INFO-F-403 – LANGUAGE THEORY AND COMPILING – PART 2

S-COBOL

Jean-Sébastien LERAT and Gilles GEERAERTS

Academic year 2013–2014

1 Introduction

During the first part of the project, you have identified the *Lexical Units* and you have implemented a JAVA lexical analyzer for S-COBOL (**S**implified **CO**mmon **B**usiness **O**riented **L**anguage).

This was the first part of your own compiler! In this second part, you will finish the compiler by adding the Parser and the Sementics Rules in order to produce the executable program.

To make your task easier, we ask you to use the LLVM toolchain. Your compiler will generate intermediate code (LLVM code) from a S-COBOL source file and the remaining tools of the LLVM toolchain will optimize your produced code and compile it natively for a specific architecture[1]. You can find more information on LLVM in the dedicated documentation. Note also that practicals number 7 and 8 will introduce you to LLVM.

2 The Grammar of S-COBOL

You will find hereunder thegrammar for S-COBOL. In this grammar, we have chosen to add the unary minus operator into the grammar (see rule 33). As you can see, the unary operator has to be distinguished from the binary operator (see rule 31 and rule 47). We have also chosen to express an **IMAGE** as a terminal **BUT** you can redefine an image because of your lexical analyzer implementation (need of information as the number of digits). Moreover, in your Sementic Analyzer, you have to check that the red (**ID**) identifiers are the same (i.e., the ID given at the begining of the code must match the ID at the end of the code). Finally, identifiers, integers and strings are written in bold.

[1]	<PROGRAM>	→	<IDENT> <ENV> <DATA> <PROC>
[2]	<IDENT>	→	identification division <END_INST> program-id. **ID**<END_INST> author. WORDS<END_INST> date-written. WORDS <END_INST>
[3]	<WORDS>	→	<WORDS> **ID**
[4]		→	**ID**
[5]	<END_INST>	→	.\n
[6]	<ENV>	→	environment division<END_INST> configuration section<END_INST> source-computer. WORDS<END_INST>

[1]Note that We will only test your code on a x8664 architecture.

			object-computer. WORDS<END_INST>
[7]	<DATA>	$\rightarrow$	data division<END_INST>
			working-storage section<END_INST> <VAR_LIST>
[8]	<VAR_LIST>	$\rightarrow$	<VAR_DECL> <VAR_LIST>
[9]		$\rightarrow$	ε
[10]	<VAR_DECL>	$\rightarrow$	<LEVEL> **ID** pic IMAGE<END_INST>
[11]		$\rightarrow$	<LEVEL> **ID** pic IMAGE value **INTEGER**<END_INST>
[12]	<LEVEL>	$\rightarrow$	**INTEGER**
[13]	<PROC>	$\rightarrow$	procedure division<END_INST> **ID** section<END_INST>
			<LABELS> end program **ID**.
[14]	<LABELS>	$\rightarrow$	<LABELS> <LABEL><END_INST> <INSTRUCTION_LIST>
[15]		$\rightarrow$	<LABEL><END_INST> <INSTRUCTION_LIST>
[16]	<LABEL>	$\rightarrow$	**ID**
[17]	<INSTRUCTION_LIST>	$\rightarrow$	<INSTRUCTION> <INSTRUCTIONLIST>
[18]		$\rightarrow$	ε
[19]	<INSTRUCTION>	$\rightarrow$	<ASSIGNATION>
[20]		$\rightarrow$	<IF>
[21]		$\rightarrow$	<CALL>
[22]		$\rightarrow$	<READ>
[23]		$\rightarrow$	<WRITE>
[24]		$\rightarrow$	stop run<END_INST>
[25]	<ASSIGNATION>	$\rightarrow$	move <EXPRESSION> to **ID**<END_INST>
[26]		$\rightarrow$	compute **ID** = <EXPRESSION><END_INST>
[27]		$\rightarrow$	add <EXPRESSION> to **ID**<END_INST>
[28]		$\rightarrow$	subtract <EXPRESSION> from **ID**<END_INST>
[29]		$\rightarrow$	multiply <ASSING_END><END_INST>
[30]		$\rightarrow$	divide <ASSING_END><END_INST>
[31]	<ASSING_END>	$\rightarrow$	<EXPRESSION>,<EXPRESSION> giving **ID**
[32]	<EXPRESSION>	$\rightarrow$	<EXPRESSION> <OP> <EXPRESSION>
[33]		$\rightarrow$	(<EXPRESSION>)
[34]		$\rightarrow$	– <EXPRESSION>
[35]		$\rightarrow$	not <EXPRESSION>
[36]		$\rightarrow$	**ID**
[37]		$\rightarrow$	**INTEGER**
[38]		$\rightarrow$	true
[39]		$\rightarrow$	false
[40]	<OP>	$\rightarrow$	=
[41]		$\rightarrow$	<
[42]		$\rightarrow$	>
[43]		$\rightarrow$	<=
[44]		$\rightarrow$	>=
[45]		$\rightarrow$	and
[46]		$\rightarrow$	or
[47]		$\rightarrow$	+
[48]		$\rightarrow$	–
[49]		$\rightarrow$	*
[50]		$\rightarrow$	/

[51]	<IF>	→	if <EXPRESSION> then <INSTRUCTION_LIST> <IF_END>
[52]	<IF_END>	→	else <INSTRUCTION_LIST> end-if
[53]		→	end-if
[54]	<CALL>	→	perform **ID** until <EXPRESSION><END_INST>
[55]		→	perform **ID**<END_INST>
[56]	<READ>	→	accept **ID**<END_INST>
[57]	<WRITE>	→	display <EXPRESSION><END_INST>
[58]		→	display **STRING**<END_INST>

You can consider boolean values as integer[2], 0 indicating the boolean *false* and all non-zero values indicate *true*. This implies that an expression like *not* 3 is accepted (and evaluates to 'true').

Recall that, as shown in th example of Figure 1, an S-COBOL 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.

```
    /Define metadata.
identification division.
   program-id. Algo-Euclide.
   author. Euclide.
   date-written. 300 BNC.
    /Define compiler parameters.
environment division.
   configuration section.
   source-computer. x8086.
   object-computer. LLVM.
    /Define our variables.
data division.
    /we define 3 variables (a, b, c).
   working-storage section.
    /s for signed.
      77 a pic s9(5).
    /9 for digit (int).
      77 b pic s9(5).
    /(5) for 5 digits.
      77 c pic s9(5).
    /code of our program.
procedure division.
   main section.
      * Euclide's Algorithm.
    /The first label is the start point.
      start.
    /read int from stdin and put it into a.
```

[2] This is the case in C, PERL, . . .

```
          accept a.
     /read int from stdin and put it into b.
          accept b.
     /call find label until b equals 0.
          perform find until b = 0.
     /write 'valeur:' on stdout.
          display 'valeur:'.
     /write the content of a on stdout.
          display a.
     /stop the program.
          stop run.
     /create a label called 'find'.
       find.
          move c to b.
     /call diff label until a is less than b.
          perform diff until a < b.
     /put the value of a into b.
          move a to b.
     /put the value of c into a.
          move c to a.
     /create a label called 'diff'.
       diff.
          * Compute a modulo b.
     /a becomes a-b.
          subtract b from a.
end program Algo-Euclide.
```

Figure 1: An example of program (the file *Euler.cbl*) written in S-COBOL

3 Priority

The priority and associativity of all operators are reported in Table 1.

Operator(s)	Associativity
`( )`	none
`not`, `-`	right
`*`, `/`	left
`+`, `-`	left
>, <, >=, <=, =	left
`and`	left
`or`	left

Table 1: Pirority (operators at the top of the table have highest priority) and associativity of operators.

4 *Desiderata*

For this project, we ask you to develop a Parser (with semantic analyzer and code generator) for S-COBOL . You should:

1. Apply the necessary algorithms (remove recursion, remove useless symbols, tranform the grammar to respect priorities, etc) to transform the given grammar into an LL(1) equivalent grammar.

2. Compute and report the LL(1) *parsing table* and the first/follow sets.

3. Decorate your grammar with semantic rules to generate the appropriate LLVM code.

4. Implement a recursive descent parser that produces LLVM code.

The implementationn will be realized using JAVA 1.6[3] and should rely either on the *Lexical Analyzer* you made for the first part of the projet, or the one that is given on the Université Virtuelle. You should hand in:

- A PDF report containing the parsing table, the first/follow sets, your transformations, and presenting your work, with all the necessary justifications, choices and hypothesis.

- The source code of your compiler.

- 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 four folders:

doc contains the JAVADOC and the PDF report;

test contains all your example files;

dist contains an executable JAR;

more contains all other files.

Your implementation[4] 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()*). Your Parser has to be a public class (called *Parser*). Your compiler should also contain the executable public class (Main) allowing to use the command:

java -jar yourJarFile.jar cobol-source-file.cbl

You will compress your folder (only zip) and you will submit it on Université Virtuelle for **Wednesday, December 18th 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. You should keep the same groups as for the first part.

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

[4]All non-standard libraries are forbidden.

5 Ressources

- DFA Simulator
- COBOL for AIX
- COBOL, notes de cours
- Tutoriel COBOL
- COBOL, un langage de programmation
- WikiVersité : COBOL
- LLVM 3.4 Reference Manual

Good work!

Bonus

You can enhance your S-COBOL compiler by adding some features. For instance, accepting a more complex *string* form (which accepts simple and double quote), real numbers,...