Université Libre de Bruxelles

Faculty of Science

Computer Science Dept.

INFO-F-403 : Introduction to Language Theory and Compilation

Project 1 : S-COBOL

Authors

Chaste Gauvain, Ooms Aurélien

Wednesday 30th October, 2013

Academic year 2013 - 2014

Abstract

COBOL (**CO**mmon **B**usiness **O**riented **L**anguage) is a a project initiated by the United States Department of Defense in order to simplify the implementation of management tools. The outcome of this project was the definition of a rather verbose language 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 is the first half of the compiler project. In this part of the project, we will deal with the syntactical analysis part of the compiler, i.e., write the lexer that splits a piece of S-COBOL into a sequence of tokens.

Contents

1 Theory

1.1 Regular Expressions

Here are shown the different special units belonging to the S-COBOL language under their regular expression (RE) form. For any keyword the RE is simply the keyword itself.

1.1.1 Identifier

```
[A-Za-z][0-9A-Za-z_\-]{0,15}
```

1.1.2 Image

```
s?9(\([1-9][0-9]*\))?(v9(\([1-9][0-9]*\))?)?
```

1.1.3 Integer

```
(\+|-)?(([1-9][0-9]*)|(0))
```

1.1.4 Real

```
(integer)(\.[0-9]+)
```

1.1.5 String

```
'[0-9A-Za-z\+\-\*/:!\? ]*'
```

1.1.6 Comment

```
(\*|/).*\.\n;
```

1.2 Finite Automaton

A simulator of the deterministic finite automaton (DFA) implemented for this assignment can be found in the dfa folder of our work. This DFA recognizes only two S-COBOL keywords. The mechanism used to recognize the rest of the keywords can easily be deducted from those two examples.

2 Implementation

2.1 Choices

In order to maintain efficiency or to avoid ambiguity we have added a certain number of constraints on the S-COBOL language. Here is the exhaustive list of our arbitrary choices and the reasons why we have made them.

2.1.1 Signed numbers vs arithmetic expressions

To avoid ambiguity an arithmetic expression will only be considered as one if a space is present between the operator and the operand. If no space can be found between the two, the token will be taken as an integer or a real.

2.1.2 Comments

Comments have to be on their own line (only space and tab characters are allowed). This way the / and * characters will not be seen as operators.

The RE for comments becomes:

```
^[ \t]*(\*|/).*\.\n;
```

2.1.3 Images vs Identifier

We chose to make the parentheses in an image mandatory. This constraint can be made without loss of generality and ensure a more intuitive system:

- Token 9 can be interpreted as an integer and is no longer cause of ambiguity.
- Tokens s9 and s9v9 can be used as identifiers while image representations s9(1) and s9(1)v9(1) still exist.

Then the RE for Images becomes:

```
s?9(\([1-9][0-9]*\))(v9(\([1-9][0-9]*\)))?
```

2.1.4 ASCII only

Our program will only recognise ASCII characters but could easily be extended to utf-8 charset.

2.1.5 Keywords in lower case

To ensure harmony our program will only accept lower case keywords. This allows the use of keywords as identifiers just by capitalizing one letter at least.

2.2 Lexer

We have implemented our lexical analyser in two different ways. One is Regex based, using compiled java patterns and the other is the direct implementation of the DFA recognizing S-COBOL.

2.2.1 Regex

The first implementation uses one big RE which is the product of the disjunction of all lexical units RE.

```
(A)|(B)|(C)|...
```

The order in which they are concatenated is chosen to make sure no ambiguity is possible.

2.2.2 DFA

A state is a class composed of a *token* (the read input so far) and one function *next*. This method is the transition function, it takes as parameters a character and another state.

A state can be final, each of the final states are bounded to a syntactical unit, a meaningful token in S-COBOL. The other ones are non-final and correspond to the rejecting states of the DFA. The DFA is then a map filled with all the states created.

The keywords and identifiers states have been automatically generated by a custom python script to avoid repetitive work.

In order to detect comments as they were defined in subsubsection 2.1.2, we had to implement two DFA's : one for the tokens standing alone on a line and the second for all the other tokens. The detail of the implementation can be looked at in the LexicalAnalyzer3 class.

2.3 Parser

The parser we implemented finds the variables and labels. To find the variables the parser detects the *data division*, variables are then all identifiers followed by images. The labels are all identifiers found in the *procedure division* that are not in the variable table.

3 Use Cases

3.1 Regular usage

stdin prompt

```
# java -jar dist/s_cobol_lexical_analysis.jar
```

stdin pipe

```
# cat test/in/1 | java -jar dist/s_cobol_lexical_analysis.jar
```

file stream

```
# java -jar dist/s_cobol_lexical_analysis.jar test/in/1
```

3.2 --mode option

--mode is one of [class, regex, map]. The default one is *class*. *map* is far from complete.

```
# java -jar dist/s_cobol_lexical_analysis.jar --mode class|regex|map
```

3.3 --comment flag

If the --comment flag is set the lexer doesn't discard comments.

```
# java -jar dist/s_cobol_lexical_analysis.jar --comment
```

3.4 Run tests

In order to run tests *bash* must be installed.

```
# python3 test/run
```