Automata & Languages Project

Course 2020-2021

3th Assignment: Syntactical parser

Before sessions of November 23rd and 29th for each group

Goal of the assignment:

The goal of the assignment is to code a parser by using the C programming language and the tool for automatically generating parsers named ***bison***. This parser will be used by the ALFA compiler under development along the course.
You should also provide the parser with a main C program to test it.

Please, follow these suggestions while you solve the assignment.

Develop of the assignment:

1. Coding an input file describing ALFA syntax for *bison*

You should write a file named ***alfa.y*** that contains a valid input file for ***bison*** that describes the syntax of the programming language ALFA.

When you process ***alfa.y*** with bison it is possible that error messages about conflicts occur. This conflicts refers to those that the pushdown automaton generated by bison encounters, For further information regarding these errors you should compile ***alfa.y*** in the following way:

bison -d -y -v alfa.y

the flag ***-v*** generates the file ***y.output*** that contains the full description of the automaton and the possible conflicts (if they exist)

2. Specification input file for *flex*

The student should modify the file *alfa.l* used in the previous deliverables for linking it with the file *alfa.y* generated by *bison*. For this purpose you should follow your teacher's suggestions.

You must change #include "tokens.h" by **#include "y.tab.h"**

3. Coding the test main program

The student should write (with the C programming language) a main program to test the parser generated by *bison* with these requirements:

- It has to be named as ***pruebaSintactico.c***.
- The corresponding executable will be invoked in the following way

pruebaSintactico <input file name> <output file name>

- The structure of both files is described in the following sections.

Remark: is very important to comply with the format of the output files because the evaluation of your code will be based on automatic testing scripts.

4. Input files description

The input file contains plain text for programs (that could contain mistakes) written in the Alfa programming language described elsewhere in the resources of the course.

5. Output files description

The output files contain two different types of lines. More specifically

- A line for each **TOKEN** shifted (matched) by the scanner.
- A line for each **PRODUCTION RULE** reduced by the parser.

Lines for shifted **TOKENS** should contain these information and in this format:

**;D:	<token>**

where

- **<token>** is the input fragment matched by the scanner for the token under consideration.
- **D:** and **<token>** are followed by a tab symbol.

Lines for reduced production rules follow this format:

**;R<# rule>:	<rule>**

where

- **<# rule>** is the number by which each rule is identified in the grammar es el número que identifica la regla que se reduce en la gramática de *ALFA*.
- **<rule>** is the text of the reduced rule just like it is in the grammar file.
- **R<# rule>** and **<rula>** have between them a single tab symbol .
- Words in the text **<rule>** have a space symbol among each other.

6. Error handling.

When a syntactic error occurs, the parser generated by ***bison*** should print in the stdout a line for each error with this format:

```
****Error sintactico en [lin <# line>, col <# character>]
```

where

- **<# line>** is the input file lien number where the last **TOKEN** matched by the scanner appears.
- **<# character>** is the column of the corresponding line where this **TOKEN** starts.

IMPO: You can Uuse the function **yyerror** for this purpose

When a morphological error happens, the scanner should print in the stdout, a message about the specific mistake. This message has to follow literally the described format. Pay special attention to not use any Spanish specific character. Otherwise the automatic part of the correction will generate poor marks for this reason.

Only two morphological errors will be cached: not allowed characters and too large identifiers. For each of them the scanner will print the message described in previous assignments.

It is important to avoid the generation of two messages for a single morphological error. Be careful to treat morphological errors only as morphological errors instead as syntactic ones too.

7. Examples

Three input files are provided in order to run initial tests of your program (*entrada_sin_1.txt, ..., entrada_sin_3.txt*). The corresponding output files are also provided (*salida_sin_1.txt, ..., salida_sin_3.txt*). Assuming your executable file is named *pruebaSintactico*, a command like:

```
pruebaSintactico entrada_sin_1.txt misalida_sin_1.txt
```

should generate a file named *misalida_sin_1.txt* equal to the file *salida_sin_1.txt*. That means that the command

```
diff -bB salida_sin_1.txt misalida_sin_1.txt
```

should generate no output.

Deliverables:

You should upload to the corresponding Moodle task only one zipped file according to these requirements:

- It should contain all the needed sources (.l, .y, .h and .c files) to solve the assignment. You don't need to include neither the *lex.yy.c* nor the *y.tab.c* files because they will be generated from the *.l* and *.y* sources.

- It should contain a *Makefile* for the *make* linux tool that for the goal ***all*** generates the executable file named ***preubaSintactico***.

- The zip file should be named

Names&Surnames_that_identify_the_group_sintactico.zip

Where

- Names&Surnames_that_identify_the_group is the one that your professor suggests.

VERY IMPORTANT REMARK: PARSERS WITH CONFLICTS WILL FAIL

It is also important **that the student remembers that these assignments depend on each other and he is responsible for ensuring to not propagate mistakes found in an assignment to the next.**