Push Down Automaton (PDA) Tester

Project. Part II

Arturo Burela

This program reads from a file the production rules that define a CFG, then it creates an equivalent PDA and tests if an input string is accepted by the automaton. Also prints all the accepted transitions.

The file must be defined as follows:

- First row: `Set of non terminal symbols`
- Second row: `Alphabet symbols`
- Third row: `Initial non terminal symbol`
- Fourth row and up: `Grammar productions`

`Set of non terminal symbols` indicates all the non terminal symbols separated by commas. No whitespace. Example:

```
S,P,t
```

`Alphabet symbols` indicates the alphabet symbols separated by commas. No whitespace. Example:

```
a,b,c,0,1
```

`Initial non terminal symbol` indicates the initial non terminal symbol. Example:

```
S
```

`Grammar productions` indicates the grammar productions in the following format: NTI->w where w is a string in (V U ∑)* . Example:

```
P->0P0|1P1
```

`Epsilon Transitions` are represented by '&' character!. For the previous functionality '&' can not be used as an element of `Alphabet symbols`.

Usage instructions

This program has been tested on Debian 9 and MacOS 10.12. It is necessary to have a C++11 compiler. There is a makefile to facilitate the process, clone the project in any desired location, enter to the project directory and type `make` in the terminal to compile the program. If you do not have make installed visit https://www.gnu.org/software/make/ (https://www.gnu.org/software/make/) . You may also compile the source yourself.

To use the program, first create the CFG file as described previously. Here is an example of a full CFG file:

```
P
0,1
P
P->0P0|1P1
P->0|1|&
```

The program receives the CFG filename as the first argument and the string to test as second argument. For example to use the CFG file 'CFG1.txt' and test string 'aab' the program must be run like this:

```
./pda CFG1.txt aab
```

The program will make the test and print the results. This approach facilitates testing different strings with different automatons.

Program Explanation

This program is written in c ++. It consists of 2 classes in 2 cpp files and a main function in a third file that makes use of them. They are divided as follows:

- main.cpp: `Main function`
- pda.cpp: `PDA Class`
- state.cpp: `State Class`

`Main function` only contains a main function that checks for two arguments and passes them to PDA class constructor. This file includes pda.cpp.

`PDA Class` defines the PDA structure. Contains all the states, the string to test, a pointer to the initial production and the paths of an accepted string. This class has the following functions:

- `test`: Calls explore recursive function on initial state, receives the paths and prints if the string is valid or not along with the paths.
- `logStates` Logs all states data.
- `logPaths` Logs all valid paths.
- `loadFile` Reads the CFG file and fills PDA structure. Including states names, states links, initial and final states.

`State Class` defines states structure. It has attributes such as name, if it is final and a vector of

'link', a struct that contains the input, stackInput, stackCondition and a pointer to the destination state. This class has the following functions:

- `addLink` Receives a link struct and adds it to the link vector.
- `getName` Returns the state's name.
- `setFinal` Marks the state as final.
- `explore` Recursive function that returns a vector of paths. Each path is conformed by a vector of links. The function receives the test string and pda stack and use them to call itself at every link it matches. If the string is empty checks if the state is final and returns the appropriate value. All the epsilon links are always called first. If the received value is valid it appends the new link to the path and returns.
- `logData` Log all the state's data.

For more details see code and comments on each file. Find this project at https://github.com/ArturoBurela/ndfa-tester (https://github.com/ArturoBurela/ndfa-tester)