Middle East Technical University Department of Computer Engineering

CENG 242

Programming Languages

Spring '2014-2015

Programming Assignment 4

Due date: 15 May 2015, 23:55

1 Regulations

1. **Programming Language:** You must code your program in C++. You are expected make sure your code compiles successfully with `g++` on department lab computers using the flags `-ansi -pedantic`.
2. **Requested Files:** You are expected to submit a set of C++ source files stated in the specifications.
3. **Submission:** Submission will be done via Moodle system. You should submit your solution under the topic Programming Assignment 4 in `http://ideavideo.ceng.metu.edu.tr/242`.
4. **Late Submission:** A penalty of $5 * day * day$ is applied.
5. **Cheating:** Cheating will result in receiving 0 from all assignments and the university regulations will be applied.
6. **Newsgroup:** You must follow the Ceng242 newsgroup for discussions and possible updates on a daily basis.
7. **Allowed Libraries:** You may include and use C++ Standard Library. Use of any other library (especially the external libraries found on the internet) is forbidden.
8. **Evaluation:** Your code will be evaluated on the Moodle system. Make sure your source code can be successfully compiled and can generate expected outputs on the sample cases provided. Note that actual evaluation will be performed on another comprehensive set of test cases.
9. **Grading:** Your program will be evaluated and graded automatically using "black-box" technique so make sure to obey the specifications.
10. **Memory-leak:** The class destructors must free all of the used heap memory. Any heap block, which is not freed at the end of the program will result in grade deduction. Please check your codes using `valgrind --leak-check=full` for memory-leaks.

2 Objectives

In this assignment you will practice the Abstraction, Inheritance and Polymorphism concepts of Object Oriented Programming.
Keywords: *OOP, polymorphism,inheritance, abstract class*

3 Problem Definiton

In this assignment, you will develop an application, the *Study Computation* application, that will help computer science students to understand the computational models Deterministic Finite State Automata (DFSA) and Deterministic Push Down Automata (DPDA) .

The application enables students to

- load computational model parameters from a text file,
- see output/result of a computational model on an input,
- see computation history (sequence of model configurations) of a computational model on an input

3.1 Application Interface

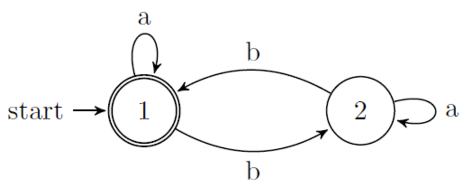
The *Study Computation* application requires two command line arguments, namely the model parameters file and the model type. The model type argument can have two values *dfsa* and *pdpa* corresponding to the DFSA and PDPA models, respectively. The application can be run with command

```
./studyComp <modelFilePath> <modelType>
```

The student can interact with the application with the following three commands:

```
r input (Print output of the model on the input)
h input (Prints configuration history of the model on the input)
q       (Quit)
```

A sample usage of the application is given in Figure 2. The application is invoked to work on the DFSA given in Figure 1. The left subfigure of Figure 1 is the content of the model parameter file for the DFSA in the right subfigure. This DFSA accepts strings that contain even number of b's.



```
2
1
1 1
2 a b
4
1 a 1
1 b 2
2 a 2
2 b 1
```

Figure 1: (left): A sample DFSA which accepts strings that contain even number of b's, (right): parameter file content for the sample DFSA

In Figure 2, interactions of the user are marked with -> signs. The application prints model parameters before interacting with user. The application prints the output of the model in response to r command and the computation history of the model in response to h command.

```
->  $./studyComp dfsa.txt dfsa
    Loading model from dfsa.txt
    Model Loaded Succesfully
    ------------------------------------------------------
    DFSA
    Number of States: 2
    Initial State: 1
    Accepting States: 1
    Input Alphabet: a b
    Transition Table:
    (1,a)->1
    (1,b)->2
    (2,a)->2
    (2,b)->1
    ------------------------------------------------------
    Please enter one of the commands to continue:
    r input (Print output of the model on the input)
    h input (Print computation history of the model on the input)
    q (quit)
    ------------------------------------------------------
->  r babab
    Input: babab
    Output: REJECT
->  r bb
    Input: bb
    Output: ACCEPT
->  h bb
    Input: bb
    Computation History:
    1 Xbb
    2 bXb
    1 bbX
    ACCEPT
->  q
    Quitting...
```

Figure 2: Sample Use of the *Study Computation* application.

3.2 Application Design

The application is designed to follow the Object Oriented Paradigm (OOP). The design is centered around two abstract classes ComputationalModel and Configuration. The application can be extended to support a new model by adding

- a new concrete subclass of ComputationalModel
- a new concrete subclass of Configuration
- minor updates to main application code.

The ComputationalModel abstract class provides interface for initializing a model instance from a text file via the *loadFromfile* method. The *print* method prints the parameters of the model. Note that each model has its own set of parameters therefore the implementation of these two methods must be model-specific. The *compute* method returns the output of a model on an input. The *computeWithTrace* method, stores the computation history in the *confList* parameter besides returning the output. Note that this parameter is passed as a reference.

```
class ComputationalModel {

  public:
    virtual ~ComputationalModel(){};// Virtual Destructor
    virtual string compute(const string& input)  = 0;
    virtual string computeWithTrace(const string& input, vector<Configuration*>& ↩
       confList) = 0;
    virtual void loadFromFile(const string& modelFilePath)=0;
    virtual void print() const =0;
};
```

State of a model during computation is named as *Configuration*. The *Configuration* abstract class represents the notion of model state. Concrete implementations vary among computational models. For instance, DPDA configuration needs to comprise stack information, on the contrary there is no stack structure in the DFSA model. The only method of this abstract class, the *print* method, prints the configuration in a model-specific format.

```
class Configuration {
  public:
    virtual ~Configuration(){};// Virtual Destructor
    virtual void print() const =0;
};
```

Owing to this design, the main application code can be simple and readable. Main function of the application creates instances of the models according to the input arguments, the *model parameter* file and the *model type*. A piece of code from the main function is given above. In this piece of code, examine the calls to *compute* and *computeWithTrace* methods.

```
    ComputationalModel* model = NULL;
    if (modelType.compare("dfsa") == 0){ model = new DFSA();}
    else if (modelType.compare("dpda") == 0) {  model = new DPDA();}
    else {cout << "Unsupported model type " << endl;}

    model->loadFromFile(modelFilePath);
    model->print();

    if (cmd.compare("r") == 0){
      cin >> input;
      cout << "Input: " << input << endl;
      cout << "Output: " << model->compute(input) << endl;

    } else if (cmd.compare("h") == 0){
      cin >> input;
      cout << "Input: " << input << endl;
      cout << "Computation History:" << endl;
      vector<Configuration*> history;
      model->computeWithTrace(input,history);
      // Printing history
      vector<Configuration* >::const_iterator it = history.begin();
      for (; it != history.end(); it++){
        (*it)->print();
        delete (*it);
      }
    }
```

4 Model Definitions

4.1 DFSA

In this assignment, we stick to the DFSA and DPDA definitions in the CENG 280 textbook. *(Lewis, H.R and Papadimitriou, C.H. Elements of the Theory of Computation (2nd ed.), Prentice-Hall, 1998.).* You can refer to the Section 2.1 of the book for more details on DFSA.

A **deterministic finite state automata (DFSA)** is a quintuple M = (K,$\sum$,δ,s,F) where

- K is a set of finite states,
- $\sum$ is the input alphabet,
- s ϵ K is the initial state,
- F $\subseteq$ K is the set of final/accepting states,
- δ, the transition function,is a function from K$\times\sum$ to K.

A configuration of a a DFSA (K,$\sum$,δ,s,F) is any element of K$\times\sum^*$. The configuration is a pair that contains the state of the machine and the unread portion of the input string.

4.2 DPDA

We present the PDA definition from the Section 3.3 of the CENG 280 textbook. DPDA is a specific case of PDA as defined in section 3.7 of the CENG 280 textbook.

A **(Non-deterministic) Pushdown Automata (PDA)** is a sextuple M = (K,$\sum$,Γ, Δ,s,F) where

- K is a set of finite states,
- $\sum$ is an alphabet (input symbols)
- Γ is an alphabet (stack symbols),
- s ϵ K is the initial state,
- F $\subseteq$ K is the set of final/accepting states,
- δ is the **transition relation**, is a finite subset of K$\times\sum$ to (K$\times(\sum\cup\{\epsilon\})\times\Gamma^*$) $\times$ ($K\times\Gamma^*$).

If $((p, a, \beta), (q, \gamma)) \in \Delta$, then M, whenever it is in state p with β at the top of the stack, may read a from the input tape (if $a = \epsilon$, then the input is not consulted), replace β by γ on the top of the stack and enter state q. Such a pair is called a **transition** of M. In a PDA, there may be several transitions of M that are simultaneously applicable at any point.

A configuration of a PDA is defined to be member of $K\times\sum^*\times\Gamma^*$. The first component is the state of the machine, the second is the unread portion of the input and the third is the content of the stack, from top to down.

DPDA: A PDA is deterministic if for each configuration there is at most one configuration that can succeed it in a computation. In other words, at any point there exists at most one applicable transition.

If the automaton reaches the configuration (f,ϵ,ϵ) -final state, end of input and empty stack- it **accepts** the input string. On the other hand, if the automaton detects a mismatch between input and stack symbols or if the input is exhausted before the stack is emptied, then it rejects.

A configuration of a PDA is **dead end** if no progress can be made starting from it towards either reading more input or reducing the height of the stack. In a dead end configuration, none of the pairs in the transition table would match the current configuration values. **A DPDA rejects its input when it reaches a dead configuration.**

5 Specifications

5.1 Source Code Organization

For this assignment you will be provided with abstract classes and the main application code in the three C++ source files ***ComputationalModel.h***, ***Configuration.h***, ***main.cpp***. You should not assume any modifications to any of these files.

You are expected to implement the classes *DFSA,DPDA,DFSAConfiguration,DPDAConfiguration*. You are expected to submit the files given in the last column of Table 5.1.

Required Class	**Base Class**	**Required Files**
DFSA	ComputationalModel	*DFSA.h, DFSA.cpp*
DPDA	ComputationalModel	*DPDA.h, DPDA.cpp*
DFSAConfiguration	ModelConfiguration	*DFSAConfiguration.h, DFSAConfiguration.cpp*
DPDAConfiguration	ModelConfiguration	*DPDAConfiguration.h, DPDAConfiguration.cpp*

Table 1: Required Files

You will be provided with a Makefile so that you can build and run your applications with the following commands:

```
$make all
$./studyComp dfsa.txt dfsa
```

5.2 Input/Output Specs

There are four critical points that should be considered for both of the models:

1. The states are named as integers in the range [1,n] where n is the number of states. In the model parameter files, n is provided to define the set of states.
2. The symbols in the input alphabets will be a $ character or lowercase letters or digits. $ character may be used by some DPDA machines.
3. The character 'E' is a reserved symbol. It stands for the ϵ symbol in the DPDA definition. Besides, the empty stack will be represented with E character in configuration histories.
4. The symbols in the stack alphabet will be letters (both uppercase and lowercase) except the aforementioned reserved characters.
5. The symbols(characters) in the stack alphabet and the input alphabet will be provided in ascending order, in the model parameter files. Your *Computational.print* implementations are expected to print the symbols in the same order.

5.2.1 Model Parameter Files

A computational model instance can be initialized with a parameter file using the *loadFromFile* method of *ComputationalModel* class. Parameter file specification for DFSA is given below. The transition function is expected to occur in lines [6,6+t]. Each line represents the mapping $(p, a) - > (q)$ of the function.

```
<number of states>
<initial state>
<number of accepting states> <list of accepted states seperated by single space>
<size of input alphabet> <list of input alphabet symbols seperated by single space>
<t:number of mappings in transition function>
<each transition mapping <p> <a> <q> in a line > ( t lines)
```

Parameter file specification for DPDA is given below. The transition relation is expected to occur in lines [7,7+t]. Each line represents the pair $((p, a, \beta), (q, \gamma))$ from the transition relation.

```
<number of states>
<initial state>
<number of accepting states> <list of accepted states seperated by single space>
<size of input alphabet> <list of input alphabet symbols seperated by single space>
<size of stack alphabet> <list of input alphabet symbols seperated by single space>
<t:number of mappings in transition function>
<each transition mapping <p> <a> <beta> <q> <gamma> in a line > ( t lines)
```

5.2.2 ComputationalModel.*print* implementations

The *print* method in concrete subclasses of the ComputationalModel class is expected to print the model parameters in a model specific format. The DFSA model implementation is expected to follow the following format:

```
DFSA
Number of States: <n>
Initial State: <s>
Accepting States: <accepted states seperated by single space, in ascending order>>
Input Alphabet: <input alphabet symbols seperated by single space, in ascending order>
Transition Table: <each transition mapping in a line below this line>
(<s1>,<a>)-><s2>
```

The DFSA transition pairs must be printed in

1. ascending order wrt *s1*
2. ascending order wrt *a*

The PDPA model implementation is expected to follow the following format:

```
DPDA
Number of States: <n>
Initial State: <s>
Accepting States: <accepted states seperated by single space, ascending order>>
Input Alphabet: <input alphabet symbols seperated by single space, ascending order>
Stack Alphabet: <stack alphabet symbols seperated by single space, ascending order>
Transition Table: <each transition mapping in a line below this line>
(<p>,<a>,<beta>)->(<q>,<gamma>)
```

The DPDA transition pairs must be printed in

1. ascending order wrt *p*
2. ascending order wrt *a*
3. ascending order wrt *beta*

5.2.3 ComputationalModel.*compute* implementations

The DFSA implementation should return the string *ACCEPT* if it accepts the parameter string and the string *REJECT* if it does not accept the string.

The DPDA implementation should return the string *ACCEPT* if it accepts the parameter string and the string *REJECT* if it does not accept the string.

5.2.4 Configuration.*print* implementations

Your *print* method implementations in the subclasses of the Configuration class, are expected to print the Configuration content in a compact form. Formal definitions of the models are given in Section 4.

The DFSA configuration should comprise the state of the machine and the position of the machine on the input string. Your *print* implementation in *DFSAConfiguration* should follow the format given below. The *X* is required to mark the position of the machine on the input. You can find examples for DFSA Configuration print format in Figure 2.

```
<state> <read portion of the input>X<unread portion of the input>
```

Your *print* implementation in *DPDAConfiguration* should follow the format given below. Note that the stack content is included differently from the DFSAConfiguration format.

```
<state> <read portion of the input>X<unread portion of the input> <stackContent ↩
    from top to bottom>
```

6 Examples

For both DFSA and DPDA, we provide sample parameter files, sample inputs and responses of the application on Moodle. The sample DPDA on Moodle accepts the language $L = \{a^n b^n \$\}$.

You can test your solution with sample cases on Moodle. In addition, you can test your solutions with the following commands.

```
$make all
$./studyComp dfsa.txt dfsa < dfsa.in > dfsa.out
$diff -q -w  dfsa.out dfsa.sln
$./studyComp dpda.txt dpda < dpda.in > dpda.out
$diff -q -w  dpda.out dpda.sln
```