PROJECT

In the project, you will convert a given grammar into a recursive-descent parser code. There are four different semantic checks you have to perform. The details of these checks are listed in the next section.

RULES OF THE PROJECT

- In the project cheating is absolutely forbidden. Definition of cheating is to give your code or part of your code to a friend, to pay money to someone to implement the project, to ask someone some information related to this project in the real world or in the internet. If you are not capable of writing your own code, your code does not belong to you.

- This is an individual project, and you cannot collaborate or work together with your friends.

- You will be given a certain number of erroneous-free and erroneous input files. Your project should process all of them and produce expected output files or error messages. For grading, your project should correctly convert error-free files. Your output C source file should be compiled with gcc correctly and executable file should be run to get a full grade. For the erroneous input files, you should display correct and **meaningful** messages. Do not write just “error” on the screen. In addition to example files, a certain number of input files will also be tested during your project demos. Thus, your code should be generic and not input-dependent.

- Your code should be compiled without any error. Projects that will not compile will not be accepted as submitted. Thus, work on your project in a step-by-step fashion and always save a version of your compilable and runnable version of your project. Do not try to write your whole project at once and try to solve all the problems at the same time. After each new update in your code try to compile and try to test it first. Do not continue adding new changes, if you are not sure the part you implemented works as it should.

- Start your project early. If you do not follow the advices given above, and if you start implementing your project late in the semester, we cannot help you immediately when you have an error in your project. First, we will ask last correctly runnable version of your code to help you. If you have thousands of lines of code and you have no idea where the error is, we simply cannot help you.

- You are also expected to write a report about your project. In the report you need to explain and list the things you do and you could not do.

- You are expected to submit your project in a zip file named as <your_name>_<your_surname>.zip. Example, Cagri_Yesil.zip. The zip file should contain
    - Makefile
    - Report.pdf
    - project.l
    - project.y

SEMANTIC ANALYSIS

Analysis 1:

The line "%rules E 3 T 1;" in the input means that non-terminal E has three distinct rules and non-terminal T has only one rule. Thus, this information must be consistent with the given grammar. In this first analysis, you have to check if there is an inconsistency between this rule definition line and the given grammar. As it is seen in the example below, the rule definition line claims that there are three rules of the non-terminal "E", but in the grammar there are only two rules. In this case, you should print an appropriate error message.

error example
%rules E 3 T 1; E -> T | T + E; T -> int

Analysis 2:

The line "%rules E 2 T 3;" in the input also means there are grammar rules with E and T non-terminals. You have to give an error message if there is no grammar rule with E and/or T. On the contrary, if a there is a grammar rule for non-terminal X but it is not defined in the rule definition line, again, you have to give an error message.

Error example 1	**Error example 2**

%rules E 3;
E -> T
  | T + E;
 	
T -> int;
	%rules E 3 T 1;
	E -> T
	  | T + E;

Analysis 3: You have to give an error message if a non-terminal is referenced but it is not defined as a rule. As it is seen in the example, non-terminal T is referenced in the second rule of non-terminal E, but it does not have any dedicated grammar rule.

Error example
%rules E 3; E -> T | T + E;

Analysis 4: You have to give an error message if there is a useless rule. As it is seen in the example below, non-terminals F and G are never referenced in the right-hand side of any grammar rule, and, thus, they are useless. Give an error message in such cases.

Error example
%rules E 3; E -> T | T + E; T -> int; F -> E + G G -> int

PROPERTIES OF THE INPUT

Your input file consist of two parts: a rule definition line and the grammar. Those parts contain some symbols and tokens. All possible tokens and symbols are given in the table below.

Each input file contains a rule definition line which indicates the non-terminals and the number of rules for each of these non-terminals. The grammar part follows the rule definition line. The grammar part contains rules and each rule starts with a non-terminal and continues with an arrow in the form of a “->” lexeme. After the rule, there can be infinite number of rules. Each rule can be a non-terminal, series of mathematical operations combined with non

Possible rules	**example**
A single non-terminal	T
A single type terminal	int
A series of non-terminals and terminals	T + E * E / int

definition	**pattern**	**example**
non-terminal	Capitalized word	E, T, START, A, B
Mathematical operations		*, +, - ,/
Or operation		|
Type reserved keyword		int, float
number	integer number	2, 3, 10, -6, +12
semicolon		;
Open parenthesis		(
Close parenthesis		)
arrow		->
Rules reserved keyword		%rules

EXAMPLE INPUT AND OUTPUT

Input

```
%rules E 2 T 3;

E -> T
  | T + E;

T -> int
  | int * T
  | ( E );
```

Output

```
#include<stdio.h>
typedef enum {INT, PLUS, TIMES, OPEN, CLOSE, END} TOKEN;
TOKEN input[20] = {OPEN, INT, CLOSE, END};
TOKEN *next = input;

int E(); int E1(); int E2(); int T(); int T1(); int T2(); int T3();

int term(TOKEN tok) {return *next++ == tok;}

int E1() {return T();}
int E2() {return T() && term(PLUS) && E();}
int E()  {TOKEN *save=next; return E1() || (next=save, E2());}

int T1() {return term(INT);}
int T2() {return term(INT) && term(TIMES) && T(); }
int T3() {return term(OPEN) && E() && term(CLOSE);}
int T() {TOKEN *save=next; return T1() || (next=save, T2()) || (next=save, T3()); }

int main(void)
{
  if (E() && term(END))
   printf ("Accept!\n");
  else
   printf ("Reject!\n");
  return 0;
}
```