Grammar to C converter

Especially in huge grammars checking input files are correct or not is really hard. You may need to make huge parse trees to solve. In our project we made a “Grammar to C converter” for solving this type of problems. Firstly, you need to write your grammar to a text file then with our lex and yacc codes we are converting them to a C file. Then for checking your inputs, you just need to write an input code, after that you can try that input in your grammar very easily. We have 4 different semantic checks we are not checking the grammar is rational or not. For example, user even can write countless times “*” symbol in his/her grammar rule. Our aim is converting the grammar correctly to a C file.

For semantic checks we are using “%rules”. In %rules line you need to define your non terminals with their rule count. For example, “%rules E 2 T 3” that means you have two non terminals one is called “E” and it has 2 rules, one is called “T” and it has 3 rules. After that you can define your rules of your non terminals.

Our semantic checks:

1. In %rules part you are determining your rule count for each non terminal .Then you are defining your rules below. Defined rules should be equal to your determined rules. For example if you wrote “%rules E 2”, E should have 2 rules otherwise it will be a error.

2. In %rules you are also determining your rules which you are going to use. If you use non terminal which is not defined in %rules. It will be a error you should define your all non terminals in %rules line. For example, if you wrote “%rules E 2” and if you use “M” in rules of “E”,it will be a error.

3. Rule count of each non terminal should be greater than 0 and also after you defined your non terminal with the rule count. You should define your rules below. Otherwise program will say “There is no rule for this nonterminal”.

4. Your every non terminal should be referenced. That means you should use your all non terminals which you defined in %rules.

Also in %rules line you are determining your non terminals in a order. Then you should define your rules of non terminals in same order. For example, if you wrote “%rules T 1 E 2 Z 3” , first you should define rule of T, then rules of E, then rules of Z.

If you pass this semantic checks program will create C file is called “grammar.c”. Then you can try your input with that C file. If you have semantic mistake program will print a error message which is relevant with your mistake.

What this project can do?

This project can do 4 semantic checks correctly and it can generate a C file which can compile (but you need to write your input before compile). I have a little problem in writing rules to relevant functions. I will give specific example for this:

%rules E 2 T 3
E -> T+ int | T + float;
T-> (T) | number | T+T;

If we assume that grammar is like that “(T)” should be the T nonterminals first rule and “number” is second rule and “T+T” is third rule. When i run the program results are like “T+T” is first function of T and “(T)” is the third function of T. But in our project order of the rules is not necessary so when code executes you can not see any difference.

Examples: We have different input examples you can find them in valid_1.zip

1- grammar
    Code generates a C code which is called grammar.c
2- invalid_1
    Code prints “Reject!”
3- invalid_2
    Code prints “Reject!”
4- invalid_3
    Code prints “Reject!”
5- invalid_4
    Code prints “Reject!”
7- valid_1
    Code prints “Accept!”
8- valid_2
    Code prints “Accept!”

9- valid_3
	Code prints “Accept!”
10- valid_4
	Code prints “Accept!”

Project Files:

README.txt => File that explain how you can run your code.
Project.l => File that defines the lexical analysis rules.
Project.y => It is a yacc file which is parsing the rules.

Tuna Ateş Koç
20150702023