CSC4180 Parsing Techniques Assignment 3

Student ID: 1200100712

Project structure (Q3)

```
The structure of my workspace:
.
├── csc4180-A3-1200100712.pdf
├── testcases
└── src
    ├── Makefile
    ├── parser.cpp
    ├── parser.hpp
    ├── compile_test.sh
    └── main.cpp
```

How to execute the parser

Uplaod the whole file to the cluster and use the following commands:

```
cd path/to/project
cd src
bash compile_test.sh
```

Then you will see 5 result files in the `./` directory

```
report
/src
/testcases
test0_result.txt
test1_result.txt
test2_result.txt
test3_result.txt
test4_result.txt
```

The format of the results

```
Start Parsing:

            Stack: $ prog
```

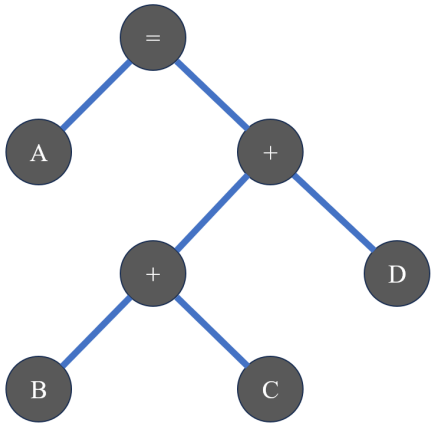
```
Processed Inputs:
   Current Input: int
            Rule: prog --> decl prog


           Stack: $ prog decl
Processed Inputs:
   Current Input: int
            Rule: decl --> fdecl


......


           Stack: $ prog
Processed Inputs: int id ( ) { var id = stringliteral ; id ( id ) ; return
intliteral ; }
   Current Input: $
            Rule: prog --> ε


           Stack: $
Processed Inputs: int id ( ) { var id = stringliteral ; id ( id ) ; return
intliteral ; }
   Current Input: $
            Rule: Match $


Accept!
```

Here `Stack` means the stack of the symbols, `Processed Inputs` means the tokens it has processed, `Current Input` means the token it is dealing with, `Rule` means the next action it will do according to the parsing table.

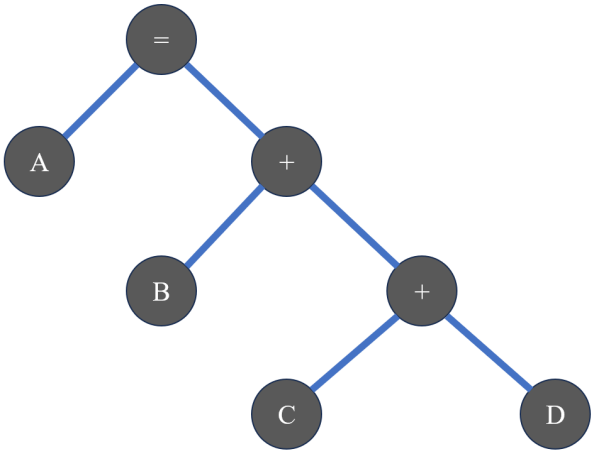
Q1. Resolve Ambiguity for Micro Language's Grammar

1. An tricky way to argue that a grammar is ambiguous is to give a counter-example input that can be parsed in at least two different ways. Can you come up with such an example for Micro's grammar to show that the grammar is ambiguous? (10%)

    1. To demonstrate the ambiguity of Micro's grammar, consider the input string "A = B + C + D". This input can be parsed in two different ways:



Parse 1



Parse 2

Here, we can have the above different parse way.

2. However, to determine whether a grammar is not ambiguous can be hard, but we can take advantage of the LL(1) parsing table.

   (a) If there are multiply defined entries in the LL(1) parsing table of the grammar, can we say this grammar must be ambiguous? Answer "Yes" or "No" and briefly explain why within 50 words. (5%)

   (b) If there is no multiply defined entry in the LL(1) parsing table of the grammar, can we say this grammar is definitedly not ambiguous? Answer "Yes" or "No" and briefly explain why within 50 words. (5%)

2.
   - (a) No. Multiple defined entries in the LL(1) parsing table indicate a conflict in the parsing process, but not necessarily ambiguity. It could be due to FIRST/FOLLOW set conflicts.
   - (b) No. Absence of multiply defined entries in the LL(1) parsing table does not guarantee the absence of ambiguity. It only indicates that the grammar can be parsed using LL(1) parsing technique, which resolves immediate leftmost derivations. Ambiguity might still exist in other parsing techniques or deeper derivations.

Q2: Simple LL(1) and LR(0) Parsing Exercises

LL(1) Grammar

Here is a simple grammar and a corresponding input string to parse.

```
E –> T | T + E
T –> int | int * T | ( E )
```

```
Input String: int + int * int
```

1. Modify the grammar to make it LL(1)
   - You can use the LL(1) web demo to see which entries in the parsing table are multiply defined.
   - Recall your answer in Q1, find out what the problem is to the grammar and how to fix it. **(Hints: left recursion elimination or left factoring?)**

From the web demo, we can find out that T and E are multiply defined.

	$	+	int	*	(	)
S			S ::= E $		S ::= E $	
E			E ::= T E ::= T + E		E ::= T E ::= T + E	
T			T ::= int T ::= int * T		T ::= (E)	

To make the grammar LL(1), we need to eliminate left factor (Here, we don't need to consider the left recursion) of the productions. Here's the modified grammar:

Original Grammar:

```
E → T | T + E
T → int | int * T | ( E )
```

Modified Grammar (eliminate left factor):

```
E → T E'
E' → + E | ε
T → int T' | ( E )
T' → * T | ε
```

Grammar for web demo:

```
E ::= T E'
E' ::= + E
E' ::= ''
T ::= int T'
T ::= ( E )
T' ::= * T
T' ::= ''
```

2. Translate your LL(1) grammar in the format that the LL(1) web demo can recognize, and then generate all intermediate tables, and parse the input string.
   - Notice that ϵ(empty string) should be denoted as two consequent single-quotes instead of one double-quotes.
   - Take a screenshot of the two tables (one is nullable, first and follow sets, and the other is parsing table) and include them in your report.

The screenshots of the two tables:

2. Nullable/First/Follow Table and Transition Table

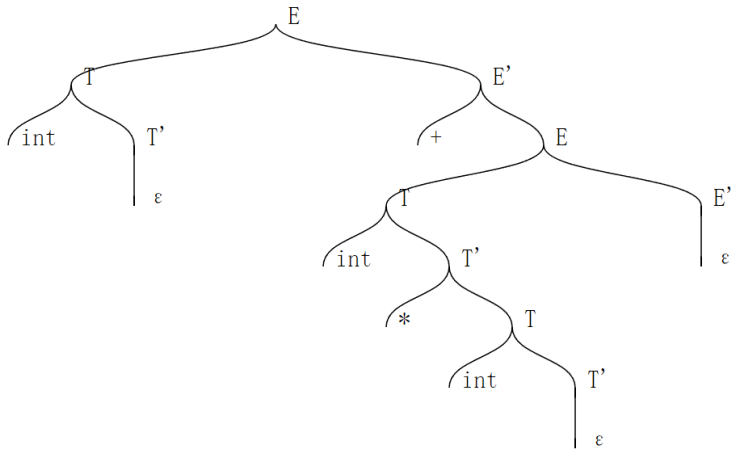
Nonterminal	Nullable?	First	Follow
S	×	int, (	
E	×	int, (	), $
E'	✓	+	), $
T	×	int, (	+,), $
T'	✓	*	+,), $

	$	+	int	(	)	*
S			S ::= E $	S ::= E $		
E			E ::= T E'	E ::= T E'		
E'	E' ::= ε	E' ::= + E			E' ::= ε	
T			T ::= int T'	T ::= (E)		
T'	T' ::= ε	T' ::= ε			T' ::= ε	T' ::= * T

3. Perform the parsing step by step in the playground to get the final parse tree.
   - Take a screenshot of the parse tree generated in the web demo, and include it in your report.

The screenshots of the parse tree:

Partial Parse Tree



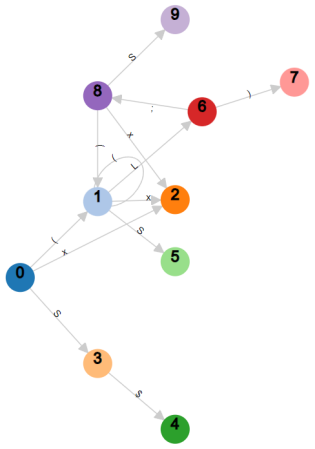
LR(0) Grammar

1. Play with the LR(0) web demo with its default grammar
    - Take a screenshot of the LR(0) automaton (both table and DFA diagram), and include it in your report.

The screenshot of the table and DFA diagram:

2. LR(0) Automaton

State	Item set
0	{S' ::= • S $, S ::= • (L), S ::= • x}
1	{S ::= (• L), L ::= • S, L ::= • L ; S, S ::= • (L), S ::= • x}
2	{S ::= x • }
3	{S' ::= S • $}
4	{S' ::= S $ • }
5	{L ::= S • }
6	{S ::= (L •), L ::= L • ; S}
7	{S ::= (L) • }
8	{L ::= L ; • S, S ::= • (L), S ::= • x}
9	{L ::= L ; S • }



2. Try in web demos and see if the default LR(0) grammar also LL(1), if the sample LL(1) grammar also LR(0)
    - Briefly explain why the LL(1) grammar is not LR(0) with the help of the LR parsing table. You need to clearly specify which conflict occurs in LR(0) parsing and why that conflict fails the LR parsing instead of telling me that there is a red row in the parsing table.

The LR(0) parsing table typically fails due to shift-reduce or reduce-reduce conflicts. In the case of the given LL(1) grammar, when constructing the LR(0) parsing table, a reduce-reduce conflict occurs.

Specifically, when constructing the LR(0) parsing table, there would be a conflict in a state where both a reduction and another reduction are possible. This happens because LR(0) parsing doesn't have lookahead, so it can't decide which action to take solely based on the current state.

In the modified LL(1) grammar:

```
E → T E'
E' → + T E' | ε
T → int T'
T' → * T | ε
```

Consider the state in LR(0) parsing when we have already seen "int + int". At this point, the LR(0) parser cannot decide whether to reduce E' to ε or T' to ε since both reductions are possible without any lookahead. This ambiguity leads to a reduce-reduce conflict in the LR(0) parsing table, rendering the grammar not LR(0).

Q3: Implement LL(1) Parser by hand for Oat v.1

How to compile and execute my program to get expected output has been attached at the beginning of the report. Here is a simple structure of my LL(1) parser class:

```
class PredictiveParser {
public:
    PredictiveParser();

    bool scan(std::string &filename);

    void addProduction(const std::string &nonTerminal, const
std::vector<std::string> &production);
    void addProduction_start(const std::string &nonTerminal, const
std::vector<std::string> &production);

    void DeBug();

    void buildParsingTable();

    void parsing();

private:
    // Data Structures
    std::string startSymbol;
    std::vector<std::string> tokens; // Input tokens
    std::unordered_map<std::string, std::vector<std::vector<std::string>>>
grammar; // Grammars
    std::set<std::string> nonTerminalSet; // The Set of nonterminals
    std::unordered_map<std::string, std::set<std::string>> allFirstSets; // First
sets
    std::unordered_map<std::string, std::set<std::string>> allFollowSets; //
Follow sets
    std::unordered_map<std::string, std::unordered_map<std::string,
std::vector<std::string>>> parsingTable; // The parsing table

    // Private Methods
    std::vector<std::string> tokenize(const std::string &source_code);
    bool isTerminal(const std::string &Input_Token);
    void calculateFirstSet();
    void calculateFollowSet();
};
```