15CSE385 – Compiler Design Lab

LL(1) Parsing

Registration Number	Name
BL.EN.U4CSE18097	R Harish
BL.EN.U4CSE18098	Raghu Ram Chadalawada
BL.EN.U4CSE18117	Siripurapu Abhishek

Overview

- Introduction
- Problem Statement
- Proposed Solution
- Technology Stack
- Design
- Implementation
- Testing & Results
- Conclusion
- References

Introduction

- **Parsing** is that phase of the compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer.
- Parser is mainly classified into 2 categories: Top-down Parser, and Bottom-up Parser.
- **LL(1)** parser is a top-down parser that uses one token lookahead

Problem Statement

To implement LL(1) parser which takes a grammar as an input, generate LL(1) parsing table and validates input string using the generated parsing table.

Proposed Solution

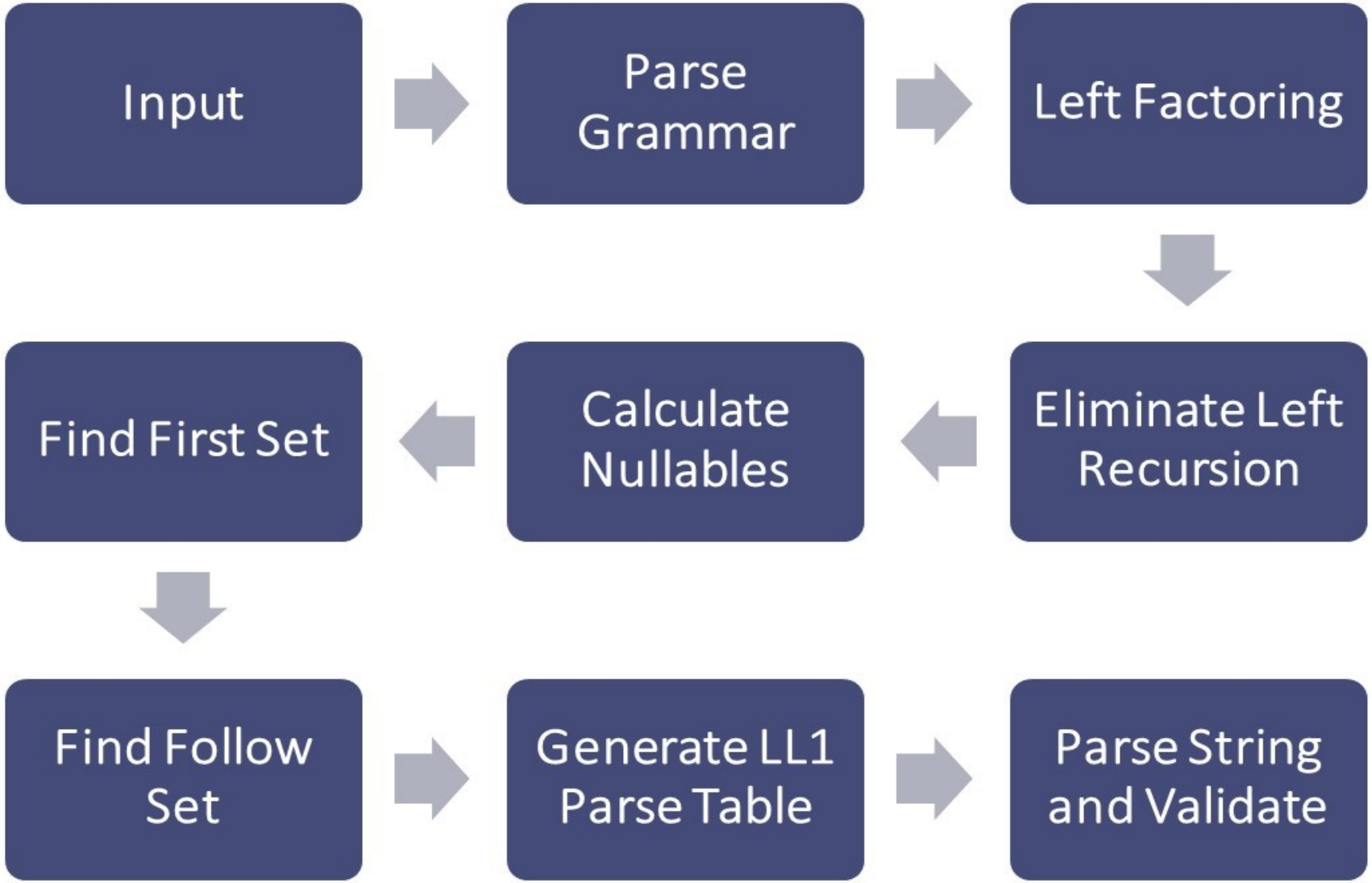
The LL(1) parser can be implemented by performing the following steps:

- Augmentation
- Eliminating left recursion (both direct and indirect)
- Appling left factoring
- Computing first and follow sets
- Constructing the parsing table
- Parsing the string using stack.

Technology Stack

- Python 3.7.0
- IDE : Google collab

Design


Input
Parse Grammar
Left Factoring
Eliminate Left Recursion
Calculate Nullables
Find First Set
Find Follow Set
Generate LL1 Parse Table
Parse String and Validate

Implementation

The implementation has been split into modules. The following are the modules we have used in the LL(1) implementation:

- **Inputs**: In this module, the context free grammar is taken as an input.
- **Parse Grammar**: This module converts the given grammar into dictionaries with non-terminals as the keys and rules as the values.
- **Left Factoring**: There are two main functionalities of this module. The first part detects the repetitive prefix and the second part performs the left factoring using the detected prefix.
- **Eliminate Left Recursion**: The left Recursion function also has two main functionalities. The first one detects for left recursion and the second part performs the left recursion removal.

Implementation

- **Calculate Nullables**: This is used to find the nullable variables in the given grammar.
- **Find First Set**: This module is used for finding the first sets for each variable by using the nullables that were found using the previous module.
- **Find Follow Set**: This module is used for finding the follow sets for each variable with the help of firsts and nullables that were found in the previous modules.
- **Generate LL(1) :** This module generates LL(1) parsing table using the first set, follow set and nullables that were found using the data that were derived by the previous modules.
- **Parse String:** The string entered will be validated using the LL(1) parsing table.
- Few other modules that help us to display the output in a readable format.

Expected System Outcomes

The algorithm is expected to take grammar as input and generate LL(1)
parse table by performing appropriate pre processing and validate if a string can be derived from the grammar and display appropriate message.

Testing and Results

```
Enter Grammar:( copy ∈ if you require lambda)
Enter $ to stop
E -> T + T | T - T | T
T -> F * F | F / F | F
F -> ( E ) | id | num
$
Entered grammar after Augmentation:
G -> E
E -> T + T | T - T | T
T -> F * F | F / F | F
F -> ( E ) | id | num
```

Testing and Results

ll1 table

	(	)	*	+	-	/	id	num	$
G	G -> E						G -> E	G -> E	
E	E -> T E'						E -> T E'	E -> T E'	
T	T -> F T'						T -> F T'	T -> F T'	
F	F -> (E)						F -> id	F -> num	
E'		E' -> ϵ		E' -> + T	E' -> - T				E' -> ϵ
T'		T' -> ϵ	T' -> * F	T' -> ϵ	T' -> ϵ	T' -> / F			T' -> ϵ

Testing and Results

```
Enter a string: id + ( id * id ) / id
+-------------+-----------------+--------------------------+
|     Rule    |      Stack      |          Input           |
+-------------+-----------------+--------------------------+
|             |        $ G      | id + ( id * id ) / id $  |
|    G -> E   |        $ E      | id + ( id * id ) / id $  |
|  E -> T E'  |      $ E' T     | id + ( id * id ) / id $  |
|  T -> F T'  |    $ E' T' F    | id + ( id * id ) / id $  |
|   F -> id   |    $ E' T' id   | id + ( id * id ) / id $  |
|  match pop  |     $ E' T'     |   + ( id * id ) / id $   |
|   T' -> ϵ   |       $ E'      |   + ( id * id ) / id $   |
|  E' -> + T  |       $ T +     |   + ( id * id ) / id $   |
|  match pop  |        $ T      |     ( id * id ) / id $   |
|  T -> F T'  |      $ T' F     |     ( id * id ) / id $   |
|  F -> ( E ) |    $ T' ) E (   |     ( id * id ) / id $   |
|  match pop  |     $ T' ) E    |      id * id ) / id $    |
|  E -> T E'  |   $ T' ) E' T   |      id * id ) / id $    |
|  T -> F T'  |  $ T' ) E' T' F |      id * id ) / id $    |
|   F -> id   | $ T' ) E' T' id |      id * id ) / id $    |
|  match pop  |   $ T' ) E' T'  |       * id ) / id $      |
|  T' -> * F  |  $ T' ) E' F *  |       * id ) / id $      |
|  match pop  |   $ T' ) E' F   |         id ) / id $      |
|   F -> id   |   $ T' ) E' id  |         id ) / id $      |
|  match pop  |    $ T' ) E'    |            ) / id $      |
|   E' -> ϵ   |       $ T' )    |            ) / id $      |
|  match pop  |        $ T'     |           / id $         |
|  T' -> / F  |       $ F /     |           / id $         |
|  match pop  |        $ F      |             id $         |
|   F -> id   |       $ id      |             id $         |
|  match pop  |         $       |              $           |
| ********** | String accepted |          ********        |
+-------------+-----------------+--------------------------+
```

Testing and Results

```
Enter a string: id + id * - id
```

Rule	Stack	Input
	$ G	id + id * - id $
G -> E	$ E	id + id * - id $
E -> T E'	$ E' T	id + id * - id $
T -> F T'	$ E' T' F	id + id * - id $
F -> id	$ E' T' id	id + id * - id $
match pop	$ E' T'	+ id * - id $
T' -> ϵ	$ E'	+ id * - id $
E' -> + T	$ T +	+ id * - id $
match pop	$ T	id * - id $
T -> F T'	$ T' F	id * - id $
F -> id	$ T' id	id * - id $
match pop	$ T'	* - id $
T' -> * F	$ F *	* - id $
match pop	$ F	- id $
**********	String not accepted	**********

Testing and Results

```
Enter Grammar:( copy ϵ if you require lambda)
Enter $ to stop
S -> S a | S b | A
A -> a A | b A | B
B -> a b | ϵ
$
Entered grammar after Augmentation:
G -> S
S -> S a | S b | A
A -> a A | b A | B
B -> a b | ϵ
```

Testing and Results

ll1 table

This grammar is not suitable for ll1 parsing

	a	b	$
G	G -> S	G -> S	G -> S
S	S -> A S''	S -> A S''	S -> A S''
A	['A -> a A', 'A -> B', 'A -> B']		
B	['B -> a b', 'B -> ϵ']		
S'	S' -> a	S' -> b	
S''	S'' -> S' S''	S'' -> S' S''	S'' -> ϵ

Conclusion

Successfully developed an algorithm that takes a grammar as an input and performs left factoring, eliminate left recursion, generate first and follow sets, LL(1) parsing table and validates if a strings can be derived using the grammar or not. The whole process is done using LL(1) parsing technique.

References

- https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/
- https://www.geeksforgeeks.org/creating-tables-with-prettytable-library-python/
- http://shorturl.at/hIQT5