BUILDING A PARSER

CS 3361-001

Spring 2022

Due Date: April 21, 2022

Instructor: Yuanlin Zhang

Bibek Dhungana

Arogya Bhatta

Kiriti Aryal

INTRODUCTION

Together, the scanner and parser for a programming language are responsible for discovering the syntactic structure of a program. This process of discovery, or syntax analysis, is a necessary first step toward translating the program into an equivalent program in the target language. The goal of this project is to build a parser, that can parse through an input text file by using scanner built in project 1, a parser that is able to identify the structure and extract the data in such a way that the output is in a form that a certain entity can understand. The scanner from the last project has been tweaked a little to facilitate the building of our parser tree. On top of grouping the tokens and taking care of the comments, this scanner outputs the value of the token. The parser is supposed to break down a given phrase or sentence into parts and describe each individual component. Then our goal is to show these components in the recursive descent pattern, also called top-down pattern. The result of this parse algorithm needs to be a parse tree made by assembling the tokens together. The parser is a language recognizer and the parse tree generated helps in recognizing the context-free language generated by the context free grammar. The parser checks the syntactics of a program, but the compiler can use its output also in the process of checking the semantic validity of the same program.

PSEUDOCODE AND ALGORITHM:

For the first part of the program, we used the same scanner built on our project. The output from the scanner is used in the second stage of the program to build the parser. These two are the first two steps of Compilation. The first one is Tokenization or Scanning (which was the primary goal of our first project.) In this project, we are using the second step of compiler/interpreter to build the parse tree.

We have the following three java classes. They are:

Main.java

This is the entry point of the program. This class takes the file name(testfile1.txt, testfile2.txt, testfile3.txt, testfile4.txt) from the terminal. And, output the parse tree using the production rule and grammar.

INPUT: filename

OUTPUT: XML based parsed tree

Note: return error if the error is encountered in the file.

Parser.java

This java file is the main class where the implementation for Parser is implemented. The following context free grammar.

```
<program> → <stmt_list> $$
<stmt_list> → <stmt> <stmt_list> | ϵ
<stmt> → id assign <expr> | read id | write <expr>
<expr> → <term> <term_tail>
<term_tail> → <add_op> <term> <term_tail> | ϵ
<term> → <factor> <fact_tail>
<fact_tail> → <mult_op> <factor> <fact_tail> | ϵ
<factor> → lparen <expr> rparen | id | number
<add_op> → plus | minus
<mult_op> → times | div
```

The input and output of this file is obvious. The input is token obtained from the scanner and the output is xml-based parse tree. The technique used is recursive descent technique. Starting from one one production rule, we can descent until completion of our project to get final output.

Input: tokens obtained from the Scanner

Output: xml based parse tree (may throw error if parsing cannot be done).

Parser(String[] tokens, String newArray[], int counter):

First, we need to copy all the tokens from to new array by removing all the null values.

We can also use '$$' to indicate the end of the program.

Since we are using recursive descent, we need to chain the function until the token is completely utilized.

The main part is:

```
Program()                    //main part of the program
        Case id, read, write: print “program”, call stmt_list()
        Case $$: match $$.       //$$ means end of the program
Stmt_list()
        Case id, read, write: print smt_list, call stmt(), call stmt_list()
        Case $$: break
        Error is encountered print the error
Stmt()
        Case id: match id, match :=, call expr()
        Case read: match read, match id
        Case write: match write, call expr()
        Error is encountered print the error
expr()                          //identify the expression by number, id and lparen
        Case id, number, lparen: call term(), call term_tail()
        Error is encountered print the error
```

Term()

Case Id, number, lparen: call factor(), call factor_tail()

Error is encountered print the error

Factor()

Case id, number: match(id or number)

Case lparen: match(( ), call expr(), match ( ))

Error is encountered print the error

Fact_tail()

Case plus, minus,rparn,id,read,write,$$: skip

Case times, div: call multi_op, factor(), fact_tail()

Error is encountered print the error

Term_tail()

Case rparen, id, read, write, $$: skip

Case plus, minus: call add_op, term(), term_tail()

Error is encountered print the error

Add_op()

Case plus, minus: match plus or minus

Error is encountered print the error

Multi_op()

Case times, div: match times or div

Error is encountered print the error

Match(string value, int index)

If valid token is encountered, print the token and increment the index by 1

The method above looks quite complicated, but we are just using production rules by using recursive descent.

DATA STRUCTURE USED

The data structures used in this project is arrays. We create one dimensional array to store the token. We have used multiple arrays to track the tokens obtained from part one of the Scanner. Again, similar arrays are used to build an xml-based parse tree. So, the main data structure used was arrays. However, we can also use Array list or list data structure for additional functionality.

OUTPUT

ACKNOWLEDGEMENT

We built our project based on the knowledge from the textbook called “Programming Language Pragmatics (4th Edition) by M. Scott, Published by Morgan Kaufmann, 2016.” Most of the methods that we have used are based on the chapters from this textbook. Bibek Dhungana and Kiriti Aryal focused on implementing the code in order to build the parser. Arogya Bhatta focused on building the report as well as working on the output. Everyone took their time to work on the pseudocode.