Kennesaw State University

College of Computing and Software Engineering

Department of Computer Science

CS 4308 Concepts of Programming Languages - Section W01

2nd Deliverable - CPL Project

Bryson Phillip bphill91@students.kennesaw.edu

3/29/2022

Problem Statement:

This writeup discusses the second stage of the CPL project where I have implemented an ADA parsing algorithm in Java. The parser is designed to interface with the outputs of the lexical analyzer, which was completed in the previous stage. The lexical analyzer tokenizes raw input code and detects if there are any unrecognizable symbols, serving as the first layer of error detection. Syntactical errors in the relationships between tokens are detectable by the parser which adheres to the specified EBNF grammar, using it to trace out a parse tree. A technique known as recursive descent is used to implement this. Each non terminal of the grammar has an associated function which detects if the following set of tokens represents a phrase of that non-terminal. Effective error statements have also been included, which refer to the specific elements within the code where a faulty statement was detected.

Summary:

This report will consist of sections discussing alterations to the EBNF grammar used, an explanation of how the code is organized, input and output files, and references. I will explain how my parser class fulfills the basic requirement of a scanner and how its processes depend on the functions of the scanner.

Detailed Description:

EBNF Grammar:

<procedure> → procedure <title> is <varlist> begin <statementlist> end <title> ;

<title> → (a b c … z A B C … Z 0 1 2 3 4 5 6 7 8 9) {(a b c … z A B C … Z 0 1 2 3 4 5 6 7 8 9)}

<varlist> → <declaration> <varlist>

| <initialization> <varlist>

| ε

<declaration> → <title> : Integer;

<initialization> → <title> := <value>;

<value> → (0 1 2 3 4 5 6 7 8 9) {(0 1 2 3 4 5 6 7 8 9)}

<statementlist> -> <varlist> <statementlist>

|<conditional> <statementlist>

| ε

<conditional> → if <condition> then <statementlist> {elsif <condition> then <statementlist>} [else <statementlist>] end if;

<condition> → <conditionB> {or <conditionB>}

<conditionB> → <comparison> {and <comparison>}

<comparison> → (<condition>)

| !(<condition>)

| <num> <operator> <num>

<num> → <title>

| <value>

<operator> → (< > <= >= /=)

Note: the ellipsis is not a symbol

This grammar describes the rules involving <procedure> which represents a most basic function within ADA. Three alterations have been made to this grammar from the last writeup. For the rules describing <varlist> and <statementlist>, an empty clause is now valid. Previously, a minimum of one item could exist within one of these lists, but now the empty string could be recognized as a complete statement. Programmatically, this creates a condition where any erroneous phrase located in place of a needed

<varlist>/<statementlist> causes the program to assume one was provided with zero elements. The erroneous phrase is then evaluated in terms of whatever would logically come after the <varlist> or <statementlist>. Additionally, <statementlist> makes use of the <varlist> instead of the <initialization>. The rule against making variable declarations in the body of a function still intends to be upheld, but will be later implemented with the use of the symbol table. Besides this, the grammar has recursive elements that allows for nested conditional statements and complex boolean expressions that preserve operator precedence and associativity, just like in the previous report.

Restrictions:

- Variables must be integers
- Variables cannot be declared and initialized in the same line
- Names are simplified
- Complex expressions cannot be assigned to variables or used in conditionals
- Functions are basic and do not include arguments
- IO is not implemented
- All of other functionalities are not present
-

Source Code Description:

The class ‘Parser’ implements the parser and is the class that includes the main function. The original lexical analyzer is used as a subroutine within the main function to generate output files and symbol tables to be used by the Parser members. The function ‘lex_main’ in the scanner outputs an array of three strings which contain the output file of tokens, a symbol table that includes the location within the file of all novel

identifiers that appear within the code, and a similar table of every lexeme and its associated position. The latter table is used to clarify where an error has occurred in error statements.

Within the parser, the function ‘find_pos’ takes in the numeric position of a token in the output file, and outputs the corresponding lexeme and its original position. The function ‘terminal_check’ will take in a possible token and check to see if it appears as the next symbol in the output. If so, it updates the cursor and prints the location of the lexeme. Each non terminal has a corresponding function that calls on functions of other non terminals to parse its substructure. At the bottom of each branch, the ‘terminal_check’ is used to determine if the correct token is present. Each operation is accompanied by a print statement that alerts the entering and exiting of a subroutine related to a particular non terminal, thus tracing out the parse tree for that particular code. The output statement from the terminal_check indicates that a particular branch of the tree has been validated. Error statements build from the ground up at a specific location. The locations of the parent code structures are appended at the front, successively, as the parser backs out of each subroutine. Finally in the implementation of <procedure> errors are printed.

Sample Code:

```
Output:
 PROC_START NAME PROC_ASSIGN FUNC_START END NAME SEMI


Symbol list:
 carson|10

Enter <procedure>
Enter terminal
PROC_START was found: procedure
Exit terminal
Enter <title>
NAME was found: carson
Exit <title>
Enter terminal
PROC_ASSIGN was found: is
Exit terminal
Enter <varlist>
Enter terminal
Exit terminal
Exit <varlist>
Enter terminal
FUNC_START was found: begin
Exit terminal
Enter <statementlist>
```

```
Enter <conditional>
Enter terminal
Exit terminal
Exit <conditional>
Enter <varlist>
Enter terminal
Exit terminal
Exit <varlist>
Exit <statementlist>
Enter terminal
END was found: end
Exit terminal
Enter terminal
NAME was found: carson
Exit terminal
Enter terminal
SEMI was found: ;
Exit terminal
```

This shows the parser output of ‘input.txt’, the most barebones possible function. Enter and Exit statements indicate the tracing of branches in a possible parse tree and ‘was

found’ indicate that a branch of a parse tree has been confirmed. This algorithm evaluates the code from left to right and there is no backtracking.

```
Output:
 PROC_START NAME PROC_ASSIGN NAME EQUAL INT FUNC_START END NAME SEMI


Symbol list:
 carson|10  var|20

Enter <procedure>
Enter terminal
PROC_START was found: procedure
Exit terminal
Enter <title>
NAME was found: carson
Exit <title>
Enter terminal
PROC_ASSIGN was found: is
Exit terminal
Enter <varlist>
Enter terminal
NAME was found: var
Exit terminal
Enter <declaration>
Enter terminal
Exit terminal
Exit <declaration>
```

```
Enter terminal
Exit terminal
Exit <declaration>
Enter <initialization>
Enter terminal
Exit terminal
Exit <initialization>
Exit <varlist>
Error in variable list at position 20:
Error at position 24 Symbol: = : Assignment operator expected
```

This shows the parser output of ‘input1.txt’. A single variable assignment was made using the incorrect operator “=” which was detected by the code.

Conclusion:

In conclusion, this implementation of a recursive descent parser has been shown to adequately trace out code that exists within this subset of the ADA grammar. It properly relies on the output of the lexical analyzer and applies further error detection capabilities.

References:

Textbook: Concepts of Programming Languages 11th Edition by Robert W. Sebesta

Website: https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html (For resources on file input streams in Java)

Website: https://www.geeksforgeeks.org/difference-between-ll-and-lr-parser/ (For resources on LL grammars)