Project: Using the Java Language to Build a Shift/Reduce Parser

Anthony S. Maida
CMPS 450

November 8, 2021

The project due date is Dec 1, 2021.

1 Introduction

This assignment is to build an SR parser using the Java language. The grammar for the parser is given below.

```
1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> ( E )
6. F -> id
```

The above grammar is written so that each rewrite rule has a unique ID. Compare this layout to that used in the "SR parser overview" lecture posted on the week of Section 4.5.1 in the textbook. We have rearranged the original grammar to remove all of the disjunctions.

One way to represent this grammar in Java is shown below.

```
String[][] grammar = {
    {"E", "->", "E", "+", "T"},
    {"E", "->", "T"},
    {"T", "->", "T", "*", "F"},
    {"T", "->", "F"},
    {"F", "->", "(", "E", ")"},
    {"F", "->", "id"}}
```

The array has six elements (each element is a slice), corresponding to the number of rules in the grammar. Java allows varying numbers of elements in the rows. For this particular grammar, the number of elements on the RHS of a rule are either one element or three elements. As explained in the “SR parser algorithm” for the week of Oct 11, we will use this information in the parser algorithm (It will be the value of r in the reduce action).

For this small program, it is convenient to let `grammar` be a global variable so that you don’t have to pass it as parameter to procedures you define.

2 Encoding the tables

An SR parser is table driven and has two tables, the `action` table and the `goto` table. We can represent each table as arrays of arrays of strings. The tables are specific to the grammar and are provided for you. The tables do not change during the parsing process, so I will refer to them as static. Here is the `action` table.

```
String[][] aTable = { // action table
    {"S5", ""  , ""  , "S4", ""   , ""      }, // 0
    {""  , "S6", ""  , ""  , ""   , "accept"}, // 1
    {""  , "R2", "S7", ""  , "R2" , "R2"    }, // 2
    {""  , "R4", "R4", ""  , "R4" , "R4"    }, // 3
    {"S5", ""  , ""  , "S4", ""   , ""      }, // 4
    {""  , "R6", "R6", ""  , "R6" , "R6"    }, // 5
    {"S5", ""  , ""  , "S4", ""   , ""      }, // 6
    {"S5", ""  , ""  , "S4", ""   , ""      }, // 7
    {""  , "S6", ""  , ""  , "S11", ""      }, // 8
    {""  , "R1", "S7", ""  , "R1" , "R1"    }, // 9
    {""  , "R3", "R3", ""  , "R3" , "R3"    }, // 10
    {""  , "R5", "R5", ""  , "R5" , "R5"    }, // 11
```

The empty string above encodes the `error` action, which means that the input is ungrammatical.

Below is a possible Java language representation of the `goto` table.

```
String[][] gTable = {
    {"1", "2", "3"}, // 0
    {"", "", ""},    // 1
    {"", "", ""},    // 2
    {"", "", ""},    // 3
    {"8", "2", "3"}, // 4
    {"", "", ""},    // 5
    {"", "9", "3"},  // 6
    {"", "", "10"},  // 7
    {"", "", ""},    // 8
    {"", "", ""},    // 9
    {"", "", ""},    // 10
    {"", "", ""},    // 11
}
```

If you access an empty string from the `goto` table, it means that there is an internal error in the parser. Just as was suggested for the `grammar` variable, it is convenient to let `aTable` and `gTable` to also be global variables.

3 The parser configuration

The parser has two supporting data structures that change dynamically during the parse. These are the input queue and the parse stack. The input queue holds the sequence of input tokens, with the last token being the ”$” string, standing for the end of the input. For this project, each token will be a string, so the input queue is a queue of strings. The parse stack is a stack of pairs (`pstackItems`), where each item consists of a grammar symbol and a state ID. The source code for the parse stack is provided to you and is found in the entry just below the turn-it-in portal for this project. You will need to implement the input queue (use the parse stack implementation as a guide).

We call the information in these data structures the parser configuration. Why? Because these are the data structures that change during each step of the parsing process. The parse tables are not part of the configuration because they do not change during the parsing process.

4 Parser control

The parsing process is built around one function called `parse1step()` which decides which of the four actions to perform in a given parse step. This function is called repeatedly until the parsing process finishes. It takes its input from global variables and does not return a value. It causes lasting effects by changing the values of global variables. This is how it would, for example, create a parse tree, by setting a global variable to point to the parse tree. Also, the input queue and parse stack will be bound to global variables and `parse1step()` changes these values. The main statement in the `parse1step()` function is a switch statement. It takes the action choice as input. There are four actions, so the switch statement will have four cases. Here is the template for the switch statement.

```
// put initializations before the switch statement
switch (choice) {
    case ACCEPT:
         break;
    case UNGRAMMATICAL:
        notGrammatical = true
        break:
    case SHIFT:             // finish defining this
        break
    case REDUCE:            // finish defining this
        break;
}
```

In the above, the variable `choice` gets its value from the value that was found in the action table. To implement `parse1step()`, implement some initializations in front of the `switch` statement. The most important initialization will be finding the action choice to be performed. Then execute the appropriate case of the `switch` statement corresponding to the action choice and return. For the cases `accept` and `ungrammatical`, there is not much to do. In my implementation, the `accept` case doesn't do anything at all. For the `ungrammatical` case, I just set a global flag indicating that the input was ungrammatical.

The main function calls `parse1step()` in a loop that terminates, either when the `accept` action is performed or the `ungrammatical` action is performed. Upon exiting the loop, the main function prints out the parse tree, assuming that the input was grammatical.

4.1 Parser output

For this project, each time `parse1step()` executes, it will print one long line of output. Sample parser output is given on Moodle at the bottom of the March 9 week. Lines 9 - 22 show the output for each call of `parse1step()`. This is for the initial input queue: `id + id * id $`. The table that is printed has several columns. They show the internal state changes of the parser as it executes the actions shift and reduce. Let's look at the columns in the table one-by-one.

One very important purpose of this table is to help you debug your parser.

Column 1 shows the parse stack before executing `parse1step()`. Similarly, the second column shows the input queue before executing `parse1step()`. The third column shows the indices to use for lookup in the `action` table. The fourth column is the value of the action choice after the table lookup. This gives us the action choice to decide which case to take in the switch statement. In the output, notice that all of the actions are either shift or reduce until the last step, which is accept. Remember that the parser terminates when the accept action is executed.

Columns 5–9 only matter when the reduce action is executed. When this happens the number associated with the "R" is a rule ID in the grammar. Column 5 shows the LHS of that rule. Column 6 shows the length of the RHS of that rule. We use this for popping the parse stack. Column 7 shows the stack that results after popping r items as dictated by the length of the RHS of the rewrite rule. It is called a temp stack because we going to push a new item on the stack before this reduce action finishes execution. Column 8 shows the indices used for lookup in the `goto` table. Column 9 shows the found value in the `goto` table, which is a state ID.

Column 10 shows the operation applied to the parse stack at the end of the action, after popping r items. Remember, for the reduce action that we have to pop r items from the parse stack corresponding to the length of the RHS. The popped items are not shown in the table.

Finally, the last column in the table shows the intermediate steps in building the parse tree. An item in this column is a stack of parse trees. When the program finishes, we print out the parse tree using indentation format, as shown in the sample output .

Remember, the table helps you debug your parser. As soon as something goes wrong with your implementation, the information will show up in the table. For the table to help you, *you have to understand the algorithm well enough to know when the information in the table is wrong*.

5 Building the parse tree

This section presents only a sketch of how to build the parse tree. Parse trees come up in two places in the output.

1. Each entry in the last column of the parser output is a stack of parse trees. To manage this information, we need a data structure to represent a parse tree and we need another data structure to represent a stack of subtrees.

2. The final output is the printed representation of the final parse tree, and that will depend on the data structure used to represent a parse tree.

Let's discuss the process of building a parse tree during the parse. Most of the work to build a parse tree happens when the `reduce` action is performed. There is only one exception to this rule. When the input processes an `id` token, then a leaf node for the parse tree is built during the `shift` action.

If you look at the parse-tree-stack column in the sample output, you can see that the parse tree is being built in a bottom-up fashion. The column label says "parse tree stack" because the stack holds subtrees. That is, the parse-tree-stack holds a stack of subtrees that eventually get assembled into a complete tree.

Here are rules for building the parse tree.

1. When implementing the `shift` action, if the the front of the input queue contains "id", then create a new tree node using the command below:

   ```
   tStack.push(new TermSym("id"))
   ```

   and push it onto the tree stack. The `termSym` data type is explained in Section 5.1.

2. When implementing the `reduce` action, there will be two cases. Either the length of the RHS will equal 1 or it will equal 3.

   First, consider the case where the length is 1, and look at the rightmost column of lines 10, 11, and 12 of the parser output. This shows the current state of the parse tree stack. Also, look at the second argument of the goto lookup value (column 8). Now we see how to build the next part of the parse tree: 1) pop the tree stack; 2) push the second argument of the look value in column 8 onto the tree that was popped; and, 3) push the new tree back onto the tree stack.

   Now, consider the case where the length of the RHS is 3. This occurs in Lines 20 and 21. We want to look at the parse tree stack in line 19 to see how it was modified to get the stack in line 20. We see that it is a stack of three subtrees as shown below.

   ```
       [F id]   [T [f id]]   [E [T [F id]]]
   ```

Now let's look at the parse tree stack for Line 20. This time it has two subtrees as shown below.

```
[T [T [F id]]  *  [F id]]   [E [T [F id]]]
```

Let's look at the changes between the Line 19 parse stack and the Line 20 stack. We encountered the binary operator "*", so we popped two items from the parse stack. Then we switched their order and inserted the "*" operator between them. This combined the two subtrees into one bigger subtree. Finally, we pushed the new subtree back onto the trees stack. The Line 21 parse tree stack is created the same we, except we use the "+" operator instead of the "*" operator. Actually, the Line 21 parse tree stack contains the final parse tree. In Line 22, we get the accept action, and as final output, we print out the single tree that is on the parse tree stack.

5.1 Data structures to support parse trees

We need data structures to support the actions just described. The data structures are needed to support two types of operations.

1. We need to represent parse trees and subtrees.
2. We need to represent a stack of subtrees. This is needed to generate the information in the last column of the output.

5.1.1 Parse trees

Let's talk about building parse trees first. To build parse trees, you must represent nodes in the parse tree. Here is an example data type to do this. It is called `TermSym` and is declared as a subclass of `ParseTree`. It directly represents a leaf node.

```
public class TermSym extends ParseTree {

    String termSym;

    public TermSym(String sym) {
        termSym = sym;  // must be +, *, (, ), id
    }

    public String toString() {
        return termSym;
    }
}
```

Here is a representation of the `ParseTree` class. This is an abstract class.

```
import java.io.PrintStream;

abstract public class ParseTree {
    PrintStream cout = System.out;

    public void printTree() { printTreeWork(0); }

    public void printTreeWork(int indentLevel) {
        String outString = "";
        for (int i = 0; i < indentLevel; i++)
            outString += "    ";
        cout.println( outString + this );
     }
}
```

The above class has two subclasses: `TermSym` (mentioned above) and `NonLeafTree` (not shown). `NonLeafTree` is used to represent `E`, `T`, and `F`. This is only one possible way to represent parse trees in Java.

If the node in the parse tree is a terminal node, then the field `termsym` will be set to either `+`, `*`, `(`, `)`, or `id`.
You will need to write operations to support the parse tree stack.

6 To get started

When you begin the project, don't worry about creating the parse tree. Just get the parser to perform the other operations. Once you are sure that the parser is performing its basic operations correctly, you can start working on building the parse tree.

For the input string to parse, keep it simple by just setting it to the value of a variable. For example, you can start with the input below.

```
    String[] inputArray = { "id", "+", "id", "*", "id"}; // grammatical input from book
    //String[] inputArray = { "id", "+", "(", "id", "+", "id", ")"};  // parens
```

The input variable is named `inputArray`. I have two copies for two possible inputs. Since I only want to use one input, I comment out the input that I don't want to use.

7 Requirements and Grading

The due date for the project is Wednesday, Dec 1, 2021. The assignment is to write an SR Parser using the Java programming language. Use the input given in the previous section (Section 6). Reproduce the output shown in "Sample parse output." The desired output is posted on Moodle for the week Oct 11-13. Specifically reproduce Line 1 and Lines 6-37. Turn in a zip file of all the files in the program, along with the output that was produced and also instructions to run the program.

Remember to think about using the sample code for building a parse stack. The code implements are parse stack and serves as a guide to implementing a tree stack data structure.

Approach the project in two stages.

1. Write the parser to generate all output, except for the parse tree. (70% of the grade)
2. Finish writing the parser so that it also generates the parse three. This involves two things: 1) printing each step in building the parse tree shown in the column titled "Parse tree stack" (20% of the grad); and, 2) printing the final parse tree in indentation format, which is shown in Lines 24-37 of the output. (10% of the grade)