COMP332 - Assignment 2

Syntax Analysis for the weBCPL language

Sophie Kaelin (45198543)

Introduction

The aim of this assignment is to create a syntactical parser to accompany the lexical parser for the weBCPL language that builds trees to visually represent the parsed code.

This report will review how my program was designed and implemented to achieve this, and will also go over how I used testing to ensure my implementation was correct.

Design and Implementation

weBCPL Statement Parsers

A variety of different statements will be parsed in a variety of ways, which is why a different implementation had to be completed for each of them. In order to determine how these statements are parsed, I had to review the structure of the context free grammar of each statement to ensure that the correct values were being piped into the correct case classes to generate the trees. Some of the statements rely on the correct implementation of other statements which meant that they couldn't be tested until every statement was created. These statements are as follows:

Unlabeled Statement

An unlabeled statement is either repeatable, iterated or a test statement. Therefore, this statement relied on the implementation of those statement parsers and would later pipe into their respective case classes.

Iterated Statement

This statement is concerned with Until, While and For statements, so depending on the values being inputted, the expressions and statements involved will be piped into their respective case classes. The Until and While statements require an expression and a statement to successfully parse. The For statements require more optional statements, so the Regex Parser opt() had to be implemented in order to return either a some or none value.

Test Statement

This statement will either have Test-Then-Else, If-Do or Unless-Do structure. This will pipe into their respective case classes.

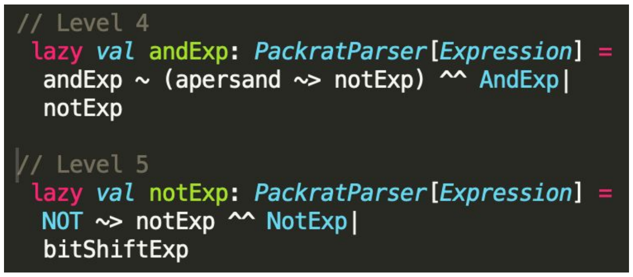
<u>Repeatable Statement</u>
A repeat statement may or may not have an expression, and that will determine whether the statement is a repeat statement, repeat while or repeat until statement. While and Until require an expression to determine how long a statement will repeat. If the statement entered does not fit any of these structures, it is passed into the simple statement.

<u>Simple Statement</u>
The simple statement is the most complex of all the statements. A simple statement will either be assigning, calling, breaking, looping, returning, finishing and end case, result, Go-to, switch on or a block statement. A Regex Parser rep1sep had to be used for the assigning statement as the context free grammar required that two expressions be passed into the case class and so the extra repeated expressions had to be caught. A simple statement can also be a Block statement, which means there is a dependency there.

<u>Block Statement</u>
There is only one structure for a Block statement which requires both rep1sep and repsep Regex Parsers.

weBCPL Expression Parsers
The expression parsers were more complex than the statements, because you have to take into consideration associativity and precedence of each one. This will determine how they are implemented and which expression will pipe to which other one. I completed the expressions in multiple sections determined by precedence: level 1-6 and then 8-12.

For each level, the context free grammar determined the correct structure required, and the other optional value was the current level above (e.g. level 1 = context free grammar OR level 2). An example of this is shown below with level 4 and 5:

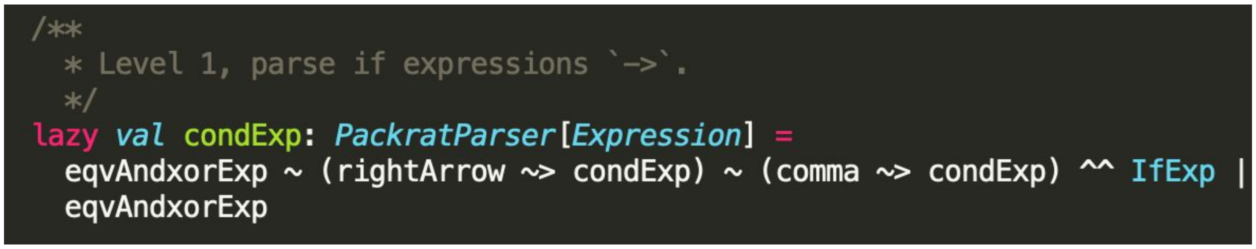

```
// Level 4
 lazy val andExp: PackratParser[Expression] =
  andExp ~ (apersand ~> notExp) ^^ AndExp|
  notExp

// Level 5
 lazy val notExp: PackratParser[Expression] =
  NOT ~> notExp ^^ NotExp|
  bitShiftExp
```

Most of these will be piped into the various case classes to be parsed.

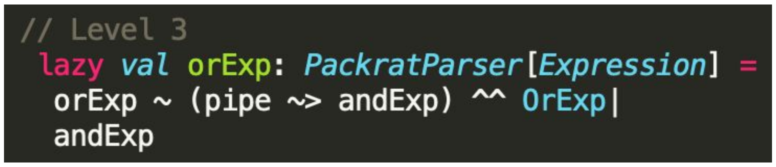
The other element to be considered for each is whether the operator has left, right or no associativity. This will determine the order of expressions, as a tree will be parsed to the left first if it is left associative so the current expression must be on the left, and vice versa if it is right associative. If there is no associativity, the expressions will only refer to themselves. Below are examples of each type of associativity:

Level 1: If Expression (Right Associativity).



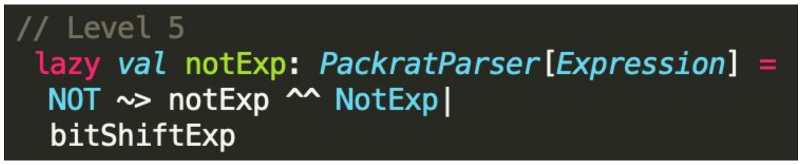
```
/**
  * Level 1, parse if expressions `->`.
  */
lazy val condExp: PackratParser[Expression] =
  eqvAndxorExp ~ (rightArrow ~> condExp) ~ (comma ~> condExp) ^^ IfExp |
  eqvAndxorExp
```

Level 3: Or Expression (Left Associativity)



```
// Level 3
 lazy val orExp: PackratParser[Expression] =
  orExp ~ (pipe ~> andExp) ^^ OrExp|
  andExp
```

Level 5: Not Expression (No Associativity)



```
// Level 5
 lazy val notExp: PackratParser[Expression] =
  NOT ~> notExp ^^ NotExp|
  bitShiftExp
```

Testing

General testing had already been implemented for the base code provided for this assignment, so the only testing that needed to be implemented was for the new statement and expression parsers. To achieve this, I had to go through each of the expressions/statements and determine a tree that would be expected for a sample input using the operators/structure that will be tested.

Unfortunately, there was limited time to complete these tests so this section is incomplete.