
# CA4003 Compiler Construction

## Assignment 1 – Lexical & Syntax Analyser

Javacc file: v3.jj

Java class: MyParser

James Nolan – 14316461

*Declaration of own work contained within WinZip directory*

*This report is not a function-by-function explanation of the program. This report discusses my observations and understandings complete with screenshots as I progressed through this assignment.*

## TABLE OF CONTENTS

# Activities & Observations

*The lexical "leading zeros" issue:*

**Activity:** In the early stages of the assignment, I ran into the issue of "leading zeros". If the user entered 00000 into the parser, it would return 5 separate integers. This concerned me as I knew this was not a valid entry. I toyed with the idea of creating an internal invalid integer token and attempted to negate this in the integer token.

**Observation:** I later realised that production rules would solve this issue. The main thing was to separate 0 from numbers 1 until 9 as "09" is an invalid integer according to the spec.

```
163 TOKEN : /*---IDENTIFIERS---*/ {
164         < INTEGER : "0" | ("-")? (["1" - "9"])(["0" - "9"])*>
```

*Fig1: Lexical no leading zeros*

*Single line comment issue:*

**Activity:** Originally, I flew through the lexical section of the assignment, however in my arrogance I made a stupid mistake. Everything after "//" was being classified as a separate token. Thank god I noticed this early on!

**Observation:** Easily fixed, originally I was going to code for strings and digits however I figured it would just be easy to break whenever a newline character was found for a single line comment.

```
72      | < SINGLE_COMMENT:  "//" (~["\n"])+ >  { System.out.println("One line comm
   ent found"); }
```

*Fig2: Single line comment definition*

*Left recursion detected ( expression & fragment )*

**Activity:** After writing up the production rules from the language specification, I compiled the jj file to be greeted with a few left recursion issues. Expression & fragment shared indirect left recursion. I know LL parsers cannot handle indirect left recursion, so this had to be fixed.

**Observation:** I applied the logic from figure 6 to re-write the left recursion by focusing in on the two offending lines and substituting in for " fragement -> expression() " with "fragement -> fragement() binary_arith_op() fragement() ". This turned indirect left recursion into left recursion which was easily dealt with. This produced a "PRIME" function ( Fig 5 ). Ìt was very important to still maintain the function of "fragment". I made a mistake here initially which I rectified later. ( see **Fig 22** ).

```
252 void expression(): {}
253 {
254         fragement() binary_arith_op() fragement()
255         | <OPEN_BRACKET> expression() <CLOSE_BRACKET>
256         | <ID> <OPEN_BRACKET> arg_list() <CLOSE_BRACKET
257         | fragement()
258 }
```

*Fig 3: expression() with an indirect left recursive issue*

```
266 void fragement(): {}
267 {
268         <ID>
269         | <MINUS_SIGN> <ID>
270         | <INTEGER>
271         | <IS_TRUE>
272         | <IS_FALSE>
273         | expression()
274 }
```

*Fig 4: fragment() with an indirect left recursive issue*

```
266 void fragement(): {}
267 {
268         (<MINUS_SIGN>)? <ID> fragement_PRIME()
269         | <INTEGER> fragement_PRIME()
270         | <IS_TRUE> fragement_PRIME()
271         | <IS_FALSE> fragement_PRIME()
272 }
273
274 void fragement_PRIME(): {}
275 {
276         (binary_arith_op() fragement() fragement_PRIME())?
277         | <OPEN_BRACKET> fragement_PRIME() <CLOSE_BRACKET>
278         | <ID> <OPEN_BRACKET> arg_list() <CLOSE_BRACKET>
279         | fragement()
280 }
```

*Fig 5: re-write of fragment to remove left recursive issue ( FRAG Prime rewritten in Fig 22 )*

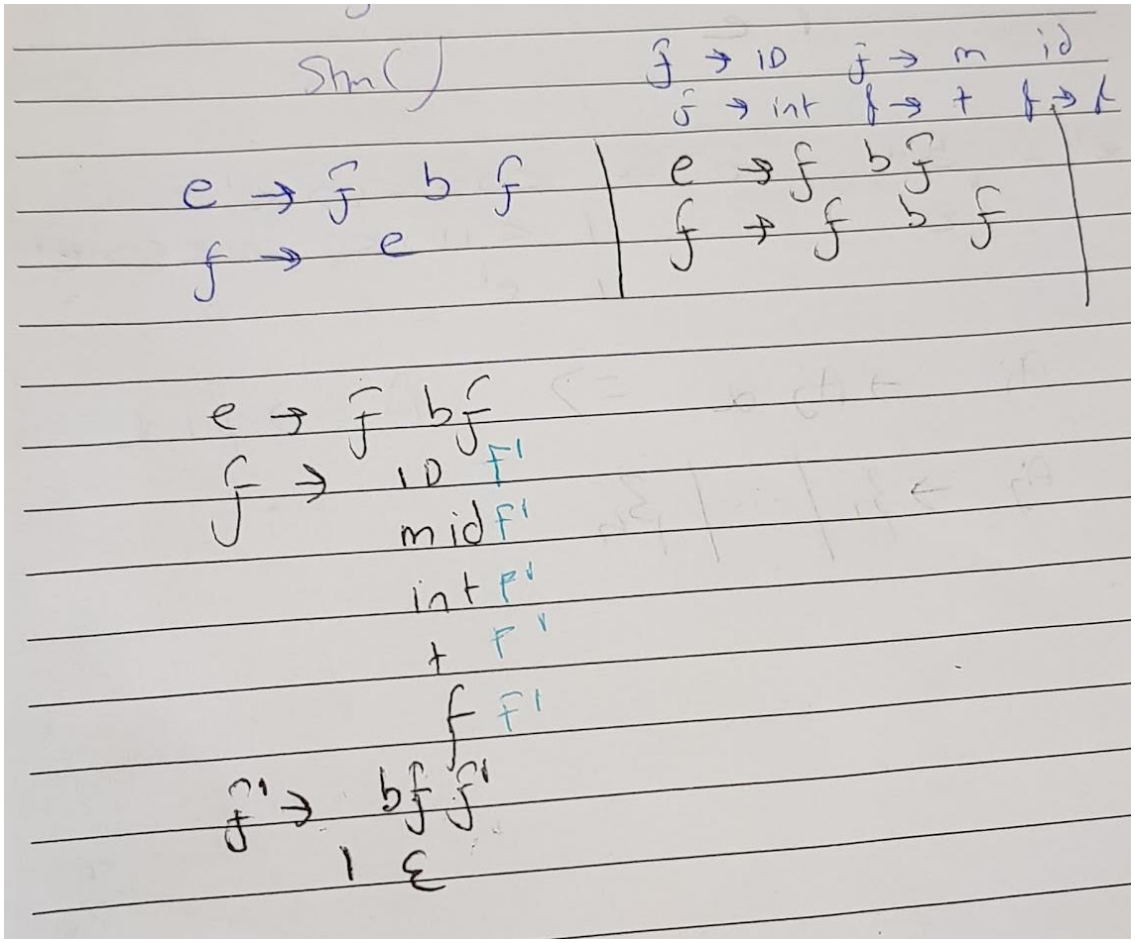

*Fig 6: Initial process I followed to remove left-factor recursion*

### *Choice conflicts I*

**Activity:** Removing the errors gave rise to many warnings. Numerous choice conflicts emerged ( Fig 7 ). There were also “empty token sequence” errors. The first choice conflict I looked at was in nemp_parameter_list(), which was simplistic.

**Observation (Empty Token Sequence) :** This was a syntax error one my part. I had a * instead of a + and due to the possibility of the line being nullable, it was always going to be chosen. This was useful to learn early in the syntax part of the assignment.

**Observation (Choice Conflict) :** Through left factoring, it was quite easy to resolve the nemp_parameter_list() issue. Whilst simplistic, this helped me understand better what choice conflicts were. The parser didn’t know which rule to apply if it encountered the token <ID> in the stream. Without this understanding early on, this assignment would’ve been much harder than it was.

```
Reading from file v3.jj . . .
Warning: Choice conflict involving two expansions at
         line 227, column 9 and line 228, column 11 respectively.
         A common prefix is: <ID> ":"
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 243, column 9 and line 244, column 11 respectively.
         A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 254, column 9 and line 256, column 11 respectively.
         A common prefix is: <ID> "("
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 256, column 11 and line 257, column 11 respectively.
         A common prefix is: <ID> "("
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Line 276, Column 9: This choice can expand to the empty token sequence and will therefore a
ys be taken in favor of the choices appearing later.
Warning: Choice conflict involving two expansions at
         line 276, column 9 and line 279, column 11 respectively.
```

*Fig 7: Post-initial recursion removal choice conflicts*

```
void nemp_parameter_list(): {}
{
        <ID> <COLUMN> type()
        | <ID> <COLUMN> type() <COMMA> nemp_parameter_list()
}
```

*Fig 8: Choice conflict v.easy*

```
void nemp_parameter_list(): {}
{
        <ID> <COLUMN> type() (<COMMA> nemp_parameter_list())?
}
```

*Fig 9: Choice conflict v.easy resolved*

### *House keeping I*

**Activity:** Some functions could be re-written in a way which did not change their purpose or introduce any new acceptable or unacceptable cases. This is something in which I did not shy away from, even though the risk of making a mistake was quite likely! ( See fig 11 & fig 12 )

**Observation (nemp_arg_list()) :** In the hopes of making the program more efficient, I attempted to replace recursion with regular expressions. In my view, this makes the program more readable also as recursion calls can be quite confusing, especially when they are unnecessary and are hard to write into an AST.

```
void nemp_arg_list(): {}
{
        <ID> ( <COMMA> nemp_arg_list() )?
}
```
#

*Fig 10: General house keeping before*

```
void nemp_arg_list(): {}
{
        (<ID> (<COMMA> <ID>)* )?
}
```

*Fig 11: General house keeping after*

```
void nemp_arg_list(): {}
{
        (<ID> (<COMMA> <ID>)* )
}
```

*Fig 12: General house keeping after mistake!*

*Choice conflicts II*

| |
|---|
| **WARNING**: Choice conflict involving two expansions at<br>line 252, column 10 and line 253, column 11 respectively.<br>A common prefix is: <ID> "("<br>Consider using a lookahead of 3 or more for earlier expansion. |
| **WARNING**: Choice conflict involving two expansions at<br>line 281, column 11 and line 282, column 11 respectively.<br>A common prefix is: "(" "("<br>Consider using a lookahead of 3 or more for earlier expansion. |

*Fig 13: Choice warnings*

**Activity:** The big choice conflicts surrounded the expression and conflict (fig 14) functions. It was time to resolve these.

**Observation (expression() housekeeping) :** First I needed to clear up expression to make this problem easier to tackle. In doing so however, I unknowingly introduced an issue in that <ID> <OPEN_BRACKET> expression() <CLOSE_BRACKET> was valid when it was not.

**Observation (Fig 16) :** Introducing a LOOKAHEAD(1) ( see fig 16 ) seemed to rid the compiler of the warning. However, as I later discovered in the assignment, when the compiler recommended a LOOKAHEAD of 3 and you gave it a LOOKAHEAD of 2, it wouldn't complain anymore, but the **issue would not be resolved!** The only way to confirm that was when you ran test cases and it failed due to the lookahead being insufficient. Thus in the fig 16 case, a lookahead ( despite it being redundant as it was 1 ) removed the warning but that was all – **issue in javacc itself!**

```
278 void condition(): {}
279 {
280         <TILDA> condition() condition_PRIME()
281         |<OPEN_BRACKET> condition() <CLOSE_BRACKET> condition_PRIME()
282         | expression() comp_op() expression() condition_PRIME() //TODO 281
283 }
```

*Fig 14: The condition issue*

```
250 void expression(): {}
251 {
252         fragment() ( binary_arith_op() fragment() )?
253         | ( <ID> )? <OPEN_BRACKET> ( arg_list() | expression() ) <CLOSE_BRACKET>
254 }
```

*Fig 15: Attempt at re-writing expression but introduced a new incorrect possibility*

```
250 void expression(): {}
251 {
252         LOOKAHEAD(1) fragment() ( binary_arith_op() fragment() )?
253         | ( <ID> )? <OPEN_BRACKET> ( arg_list() | expression() ) <CLOSE_BRACKET>
254 }
```

*Fig 16: LOOKAHEAD resolved the error? Of course not!*

### *Choice conflicts III*

**Activity:** Lookaheads are bad and I didn't want to include any of them in my assignment unless necessary. I investigated further to see what was causing the conflict.

**Observation (Fig 17) :** Working through the output of my expression() and fragment functions, I derived where < ID > was causing an issue.

**Observation (Fig 18 & Fig 19) :** I could see that through the fragment call, the parser couldn't choose whether to choose the production rule beginning with < ID > in fragment or the one in expression. Here I learned that choice conflicts weren't always so easy to spot.

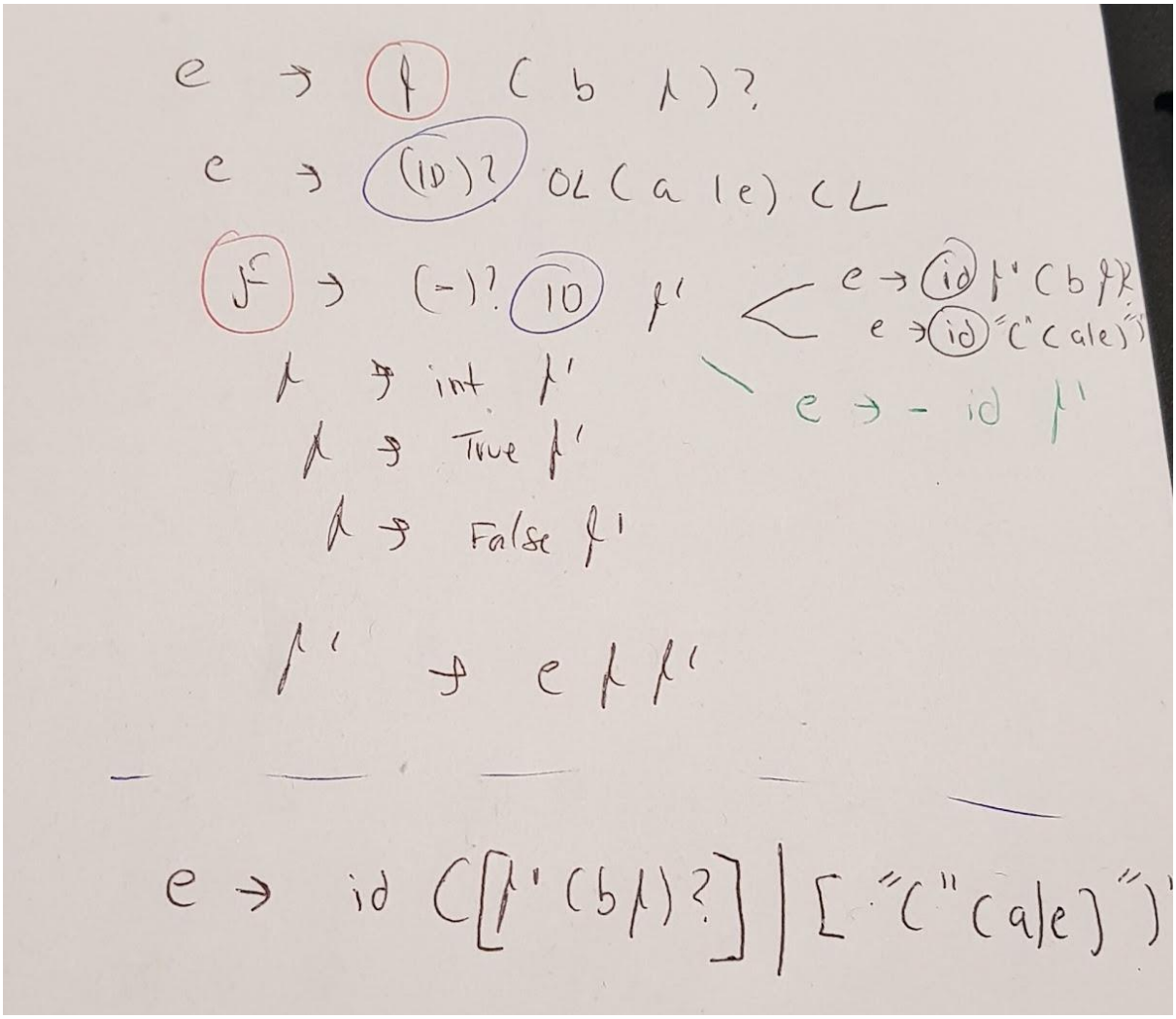

*Fig 17: What is causing the choice conflict?*

```
250 void expression(): {} //TODO FRAG CALL:: REPLACING ID
251 {
252         fragment() ( binary_arith_op() fragment() )?
253         | <ID> <OPEN_BRACKET> ( arg_list() | expression() ) <CLOSE_BRACKET>
254         | <OPEN_BRACKET> ( arg_list() | expression() ) <CLOSE_BRACKET>
255 }
```

*Fig 18: expression with line 253 causing a conflict issue through line 252*

```
264 void fragment(): {}
265 {
266         <MINUS_SIGN> <ID> fragment_PRIME()
267         | <ID> fragment_PRIME()
268         | <INTEGER> fragment_PRIME()
269         | <IS_TRUE> fragment_PRIME()
270         | <IS_FALSE> fragment_PRIME()
271 }
```

*Fig 19: line 267 causing a conflict*

*Choice conflicts IV*

**Activity:** Trying to address the problem through editing the fragment_prime, expression and fragment function made little progress. Instead, I ended up creating three conflicts.

**Observation :** This wasn't going to be easy!!

| |
|---|
| **WARNING**: Choice conflict in [...] construct at line 252, column 35.<br>Expansion nested within construct and expansion following construct<br>have common prefixes, one of which is: "-"<br>Consider using a lookahead of 2 or more for nested expansion |
| **WARNING**: Line 252, Column 91: This choice can expand to the empty token sequence<br>and will therefore always be taken in favor of the choices appearing later. |
| **WARNING**: Choice conflict involving two expansions at<br>line 252, column 91 and line 252, column 104 respectively.<br>A common prefix is: <ID> |

*Fig 20: Three more choice conflicts*

**Activity:** First I focused on the new "-" conflict.

**Observation :** The conflict was being caused again by fragement/expression. Writing the logic of the offending fragment line into expression solved the conflict issue.

```
250 void expression(): {}
251 {
252         <ID> ( ( fragment_PRIME() (binary_arith_op() fragment() )? ) | ( <OPEN_BRACKET> ( arg_list() | expression(
    ) ) <CLOSE_BRACKET> )  )
253 }
254
255 void binary_arith_op(): {}
256 {
257         <PLUS_SIGN>
258         | <MINUS_SIGN>
259 }
260
261 void fragment(): {}
262 {
263         (<MINUS_SIGN>)?<ID> fragment_PRIME()
264         | <INTEGER> fragment_PRIME()
265         | <IS_TRUE> fragment_PRIME()
266         | <IS_FALSE> fragment_PRIME()
267 }
```

*Fig 21: Expression with "-" conflict*

```
250 void expression(): {}
251 {
252         (<MINUS_SIGN>)? <ID> ( ( fragment_PRIME() (binary_arith_op() fragment() )? ) | ( <OPEN_BRACKET> ( arg_list
    () | expression() ) <CLOSE_BRACKET> )  )
253 }
254
255 void binary_arith_op(): {}
256 {
257         <PLUS_SIGN>
258         | <MINUS_SIGN>
259 }
260
261 void fragment(): {}
262 {
263         <INTEGER> fragment_PRIME()
264         | <IS_TRUE> fragment_PRIME()
265         | <IS_FALSE> fragment_PRIME()
266 }
267
268 void fragment_PRIME(): {}
269 {
270         (binary_arith_op() fragment())*
271 }
272
```

*Fig 22: Fixed the "-" conflict issue with expression ( Frag Prime re-written from **Fig 5** )*

*House keeping II*

**Activity:** I noticed this code could be cleaned up. The issue with the <ID> conflict was not going away so I understood that fragment need to be re-thought without loosing it's functionality.

**Observation :** Re-writes were dangerous. No guarantee I'd capture everything in the same way. Was able to remove fragment and fragment prime completely. The <ID> conflict remained.

```
250 void expression(): {}
251 {
252         (<MINUS_SIGN>)? <ID> ( (binary_arith_op() fragment() )* ) | ( <OPEN_BRACKET> ( arg_list() | expression() )
    <CLOSE_BRACKET> )  )
253 }
254
255 void binary_arith_op(): {}
256 {
257         <PLUS_SIGN>
258         | <MINUS_SIGN>
259 }
260
261 void fragment(): {}
262 {
263         <INTEGER> (binary_arith_op() fragment())*
264         | <IS_TRUE> (binary_arith_op() fragment())*
265         | <IS_FALSE> (binary_arith_op() fragment())*
266 }
```

*Fig 23: Fragement_PRIME removed completely*

```
250 void expression(): {}
251 {
252         (<MINUS_SIGN>)? <ID> ( (binary_arith_op() fragment() )* ) | ( <OPEN_BRACKET> ( arg_list() | expression() )
     <CLOSE_BRACKET> )  )
253 }
254
255 void binary_arith_op(): {}
256 {
257         <PLUS_SIGN>
258         | <MINUS_SIGN>
259 }
260
261 void fragment(): {}
262 {
263         <INTEGER>
264         | <IS_TRUE>
265         | <IS_FALSE>
266 }
```

*Fig 24: Fragment cleaned up, expression carrying out frag_PRIME work*

```
void expression(): {}
{
        /*( ( (
                        ( (<MINUS_SIGN>)? <ID> )
                        |  <INTEGER>
                        |  <IS_TRUE>
                        |  <IS_FALSE> )
                binary_arith_op() )
        +)
        ((binary_arith_op()
                ( (
                        ( ( ( <MINUS_SIGN> )? <ID> ) | ( <INTEGER> ) | ( <IS_TRUE> ) | ( <IS_FALSE> ) ) binary_arith_op() )*
        ))?) */ //deleted fragment binary_op fragment and replaced with this

        (
                (
                        (<MINUS_SIGN>)? <ID> | <INTEGER> | <IS_TRUE> | <IS_FALSE>
                )
                (
                        binary_arith_op()
                        (
                                (<MINUS_SIGN>)? <ID> | <INTEGER> | <IS_TRUE> | <IS_FALSE>
                        )
                )+
        )

        | ( <ID> <OPEN_BRACKET> arg_list() | <OPEN_BRACKET> expression() ) <CLOSE_BRACKET>
}
```

*Fig 25: Expression re-write absorbing fragment function*

*Understanding the choice conflicts*

**Activity:** The re-write was unsuccessful, but on paper it should've been.

**Observation :** Re-worked expression using fig 27 and fig 28 however little did I know that I had made a number of mistakes! This removed the <ID> conflict. I learnt how to tackle conflicts, by using left factoring and making the parser use that token and follow the new production rules which should be logically equivalent to the old ones.

```
void expression(): {}
{
        (
                (
                        <MINUS_SIGN> <ID> | <INTEGER> | <IS_TRUE> | <IS_FALSE>
                )
                (
                        binary_arith_op()
                        (
                                (<MINUS_SIGN>)? <ID> | <INTEGER> | <IS_TRUE> | <IS_FALSE>
                        )
                )+
        )
        | <ID>
                (//start OR
                        (
                                <OPEN_BRACKET> arg_list() <CLOSE_BRACKET>
                        )
                        |
                        (
                                (
                                <INTEGER> | <IS_TRUE> | <IS_FALSE>
                                )
                                ( binary_arith_op()
                                        (
                                                (<MINUS_SIGN>)? <ID> | <INTEGER> | <IS_TRUE> | <IS_FALSE>
                                        )
                                )+
                        )
                 )
        | <OPEN_BRACKET> expression() <CLOSE_BRACKET>
}
```

*Fig 26: Reworked expression*

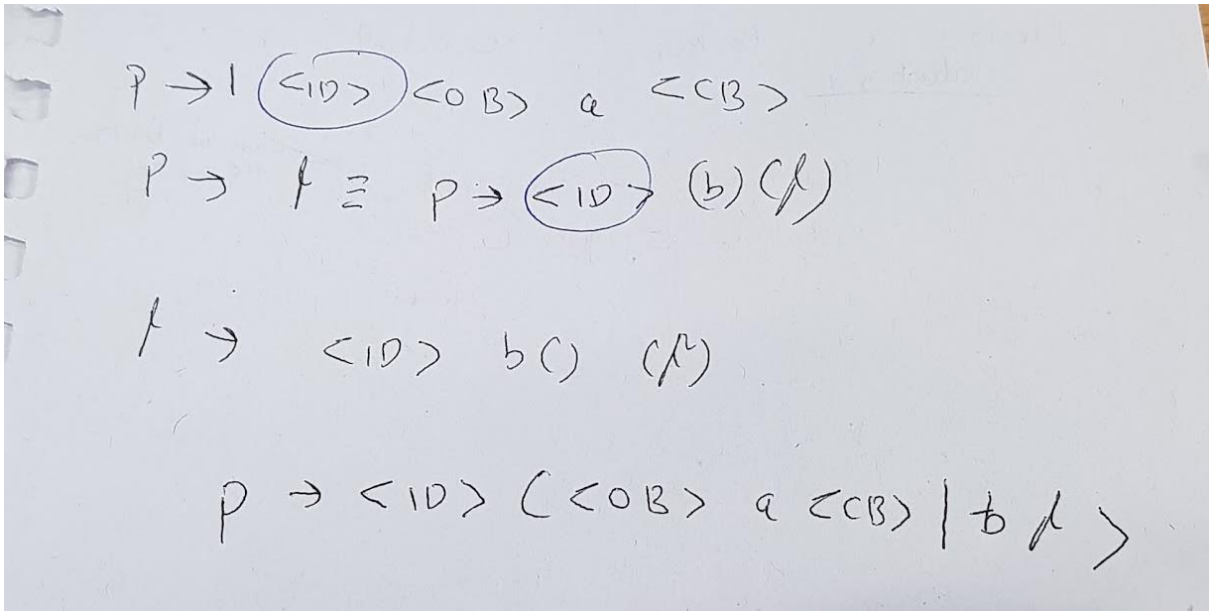


*Fig 27: Workings I*

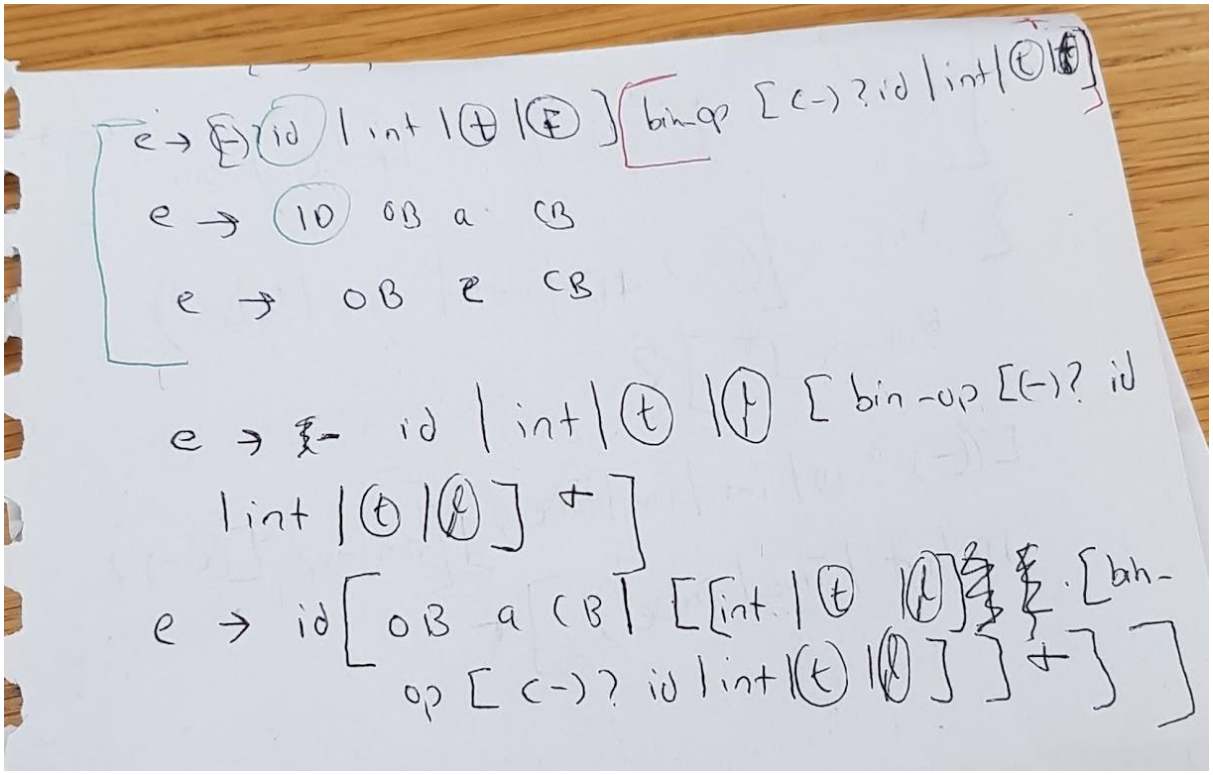


*Fig 28: Workings II*

*Choice conflicts V*

| |
|---|
| **WARNING**: Choice conflict involving two expansions at<br>line 323, column 11 and line 324, column 11 respectively.<br>A common prefix is: "(" "("<br>Consider using a lookahead of 3 or more for earlier expansion. |
| **WARNING**: Choice conflict in [...] construct at line 329, column 9.<br>Expansion nested within construct and expansion following construct<br>have common prefixes, one of which is: "\|"<br>Consider using a lookahead of 2 or more for nested expansion. |

*Fig 29: Choice conflicts*

**Activity:** Take the same approach as to the last conflict. Re-write and left factor and ensure that logical equivalence is maintained.

**Observation :** I successfully managed to remove the OR conflict with re-writing, but I could not remove the “((“ conflict. I had to use a LOOKAHEAD here to do this.

# Test Cases

My test cases spotted a lot of issues, especially in the expression function. It also spotted some issues in the way I had written from the language spec some of the easier functions. Without having tested the javacc code, I would've submitted something which basically was useless despite having done the amount of work I have for this assignment!

The test cases I ran made me introduce a second Lookahead of 3 where I had successfully removed an OR conflict from fig 29. This was because my logic was wrong.

The test cases I have will come in handy for the JJTree assignment.

# Discoveries

I discovered how better to graph what was going on, both with trees and tacking the problems on paper ( despite making many mistakes along the way! ).

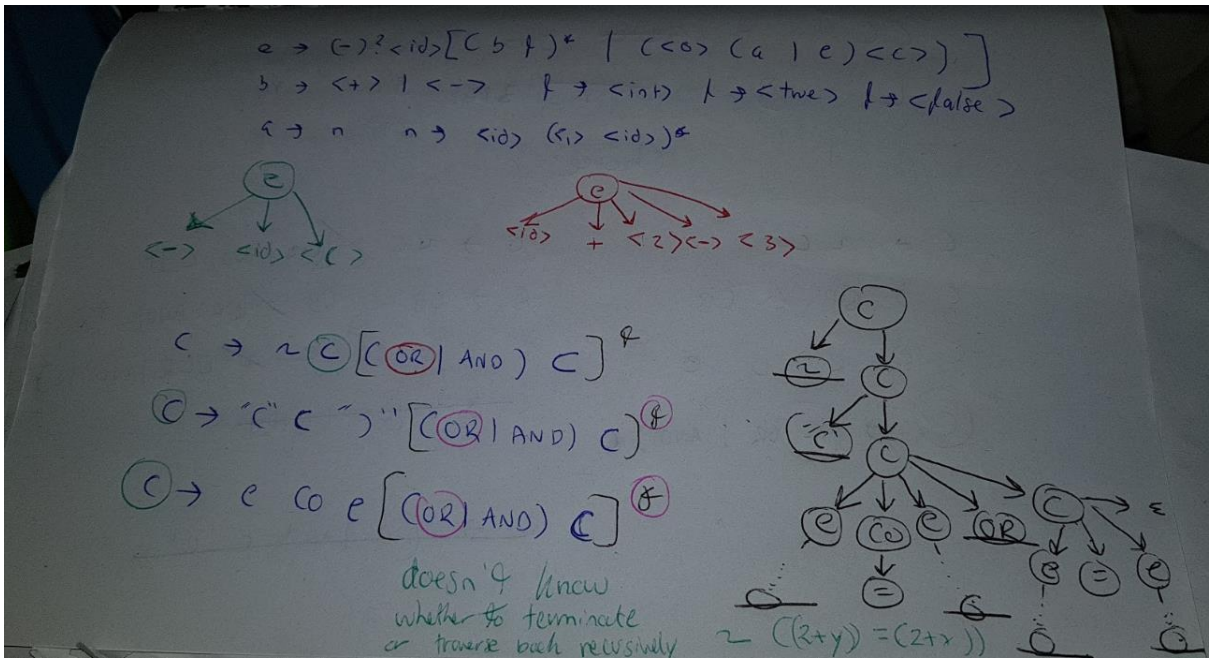


*Fig 30: Some AST's and workings or the expression condition issue*

Any mistakes made during any sort of re-write really trickled their way through. It took an entire day to fix some of the issues caused by slight mistakes in the expression re-write.

Conflicts can be resolved without lookaheads by re-writing functions whilst maintaining the logical equivalence and using left factoring to remove the offending token. The "((" conflict caused in condition is a really complex one as it conflicts with expression. The way I would address this is to write condition into expression in a way that ensures the logical equivalence is maintained.

I discovered how indirect left recursion can be tackled by translating it to left recursion, removing that recursion and applying left factoring however the process gives rise to conflicts.

That the […]* and […]? can cause nullable warnings and the parser chooses null in some cases 100% of the time.

That regular expressions can replace recursion in some cases.