CA4003 Assignment One - Conor Flynn

Introduction

I found this assignment pretty difficult and ended up taking multiple different paths before deciding on this approach. All of the different ways I tried to make the grammar LL(1) are in different branches on the Github repository for the assignment:
https://github.com/conor-f/CA4003

The real sticking point were the choice conflicts that appeared after removing the left recursion from expression and fragment. My final solution is not that great as it limits the language to only allow a simple expression in condition. The language loses very little functionality this way but is still slightly different from the original grammar. The place where this change was is clearly commented in the Parser.jj file at line 266.

I had the the clearest view of what I was doing and what edits I needed to make when I was working it out on paper, I include pictures of my workings at the end of the document for reference.

Initial Steps

The first thing I did was set up this repository and write out the test cases included in the PDF. I then made a bash script which would run my parser against all these test cases. This helped me greatly throughout the assignment. I then created all the tokens and solved the basic issues like making the language case insensitive and allowing nested comments. It was then just a matter of writing out all the production rules as they were described in the grammar and seeing what problems arose.

First Issues

Doing this presented me with two left-recursion errors. One direct (condition -> condition) and one indirect (expression -> fragment -> expression). I solved the direct left recursion easily with the help of the notes and then made the indirect left recursion direct and solved that. Removing the left recursion between expression and fragment lead to me creating fragment_prime which also had issues.

The issue with fragment_prime was that it presented a choice conflict from two possible

rules leading to epsilon. I noticed that this would not be the case if I split fragment_prime into two different rules - fragment_prime_bool and fragment_prime_num. This approach looked like it would work until I realized that fragment_prime was referenced in the general fragment rule where it was preceded by expression or a function call - neither of which we can know the type of at this stage of the compilation process, so this approach wasn't going to work.

Redo

I decided to start from scratch again, before I had removed any left recursion. I wanted to try squash the production rules for expression and fragment into one rule, and after doing this the indirect left recursion solved itself. I then removed the recursion in condition as before and got to the stage of choice conflicts. Some of these were easily removed with left factoring, but there was one final one I couldn't remove - it was in condition with a choice conflict on the <LB> token. The <LB> token could come from condition or expression and when I tried to left factor it I was just presented with the same choice conflict one level deeper. It was clear that repeating this process wasn't going to solve the conflict, so I decided to impose a limit on the language - only allowing a "simple" expression be part of a condition. This allowed me to left factor and not get any more issues so the grammar was then LL(1) and it was a good day.

Conclusion

The issue I had could have been solved by a LOOKAHEAD of 3, but it could also have been solved by using a syntactic lookahead which is probably what I would have done if I was doing this independently of the condition that the grammar had to be strictly LL(1). I sometimes found that forcing it to be LL(1) with the choice conflicts made the grammar less readable to me and I think it would be an interesting project to write a Prolog script to take a grammar in EBNF form and return a LL(1) equivalent if one exists. It could be used to keep the readability of a grammar allowing lookaheads while still getting the memory improvements of having a LL(1) parser.

Images

(2) condition() → <TILDE> Condition()
| <LB> Condition() <RB>
| expression() compOpp() expression()
~~| condition() (<OR> | <AND>) condition()~~
| Condition() orAndRule() Condition()

SIMPLIFIED:

~~A → a A~~
~~| B A~~

A → a A
| b A C
| B C B
| A D A

BECOMES:

~~A → aA~~
~~A → bAC~~
~~A →~~

A → (a A) A'
| (b AC) A'
| (BCB) A'
A' → (DA) A'
| ε

WHERE:

A = condition()
B = expression()
C = compOpp()
D = orAndRule()

a = <TILDE>
b = <LB>
C = <RB>

CONDITION LEFT RECURSION SOLVED

(1) expression → fragment → expression

expression() → fragment() binaryArithOp() fragment()
| <LB> expression() <RB>
| <ID> <LB> argList() <RB>
| fragment()

fragment() → <ID>
| <MINUS> <ID>
| <NUM>
| <TRUE>
| <FALSE>
| expression()

SIMPLIFIED:

~~expr~~ A → B C B
→ a A b
→ C a D b
→ B

B → c
→ d c
→ e
→ f
→ g
→ A

WHERE:

A = expression()
B = fragment()
C = binaryArithOp()
D = argList()

a = <LB>
b = <RB>
c = <ID>
d = <MINUS>
e = <NUM>
f = <TRUE>
g = <FALSE>

$A \to B\,B_1$
$\to aAb$
$\to caDb$

$B_1 \to CB$
$\to \varepsilon$

$B \to B_2$
$\to A$

$B_2 \to c \mid dc \mid e \mid f \mid g$

$\to$

$B \to B\,B_1$
$\to aAb$
$\to caDb$
$\to B_2$

$\to$

$B \to aAb\,B'$
$\to caDb\,B'$
$\to B_2\,B'$

$B' \to B_1\,B'$
$\to \varepsilon$

$A \to B\,B_1$
$\to aAb$
$\to caDb$

$B_1 \to CB$
$\to \varepsilon$

$B_2 \to c \mid dc \mid e \mid f \mid g$

$B \to aAb\,B'$
$\to caDb\,B'$
$\to B_2\,B'$

FRAGMENT LEFT RECURSIO
REMOVED.

(3)

Left Recursion detected: fragment_prime() ... → fragment_prime

fragment_prime() → fragment_1() fragment_prime()
→ ε

fragment_1() → binary_arith_op() fragment()
→ ε

binary_arith_op() → <PLUS>
→ <MINUS>

fragment() → <LB> expression() <RB> fragment_prime()
→ <ID> <LB> argList() <RB> fragment_prime()
→ fragment_2() fragment_prime()

fragment_2() → <ID> | <MINUS> <ID> | <NUM>
→ <ID> | <TRUE> | <FALSE>

Problem occurs as fragment_prime's ~~can go to fra~~
FIRST contains fragment_1 and ε and fragment_1's FIRST contains binary_arith_op and ε. This leads to an infinite Loop.

In fragment() we can force the → fragment_2() fragmen rule down two paths depending on if fragment_2() is possibly boolean or not:

fragment() → (<ID> | <MINUS> <ID> | <NUM>)

frag_prime_num() → ε
→ binary_arith_op() fragment()

frag_prime_bool() → ε

So fragment() → <LB> expression() <RB> fragment_prime()
→ <ID> <LB> argList() <RB> fragment_prim
→ (<ID> | <MINUS> <ID> | <NUM>) frag_prime_nu
→ (<ID> | <TRUE> | <FALSE>) frag_~~bool~~ prime_bool(

But the first rule, <LB> expression() <RB> fragment_p
or the second cannot be reduced this way as an expression or function call can (at this stage of the compilation process - before the semantic analyser) be e numeric or boolean. Therefore we cannot remove the fragment_prime() rule.

We can squash fragment_prime() and fragment_1 to deal with the issue:

fragment_1_and_prime() → ε
→ binary_arith_op() fragme
fragment_1_and_prime(

Choice conflicts (8):

nempParamList() → <ID> <COLON> type()
→ <ID> <COLON> type() <COMMA> nempPara

↓

nempParamList() → <ID> <COLON> type() (<COMMA>
<ID> <COLON> type())*

Statement_id (29d3933)

nempArgList (9673f1a)

fragment_id (6a49a1b)

Choice conflict ~ common prefix: "("

expression → fragment() fragment_1_and_prime()
→ <LB> expression() <RB>
→ <ID> <LB> argList() <RB>

fragment → <LB> expression() <RB> fragment_1_
→ <ID> fragment_id()
→ fragment_2_without_id() fragment_1_an

fragment_1_and_prime → (bin_arith_op() fragment() frag

expression and fragment both contain:
<LB> expression <RB>
We can remove that rule from fragment and fragment_1_and_prime() optional in expression()
(1736492)

Similar with <ID> conflict...

try2: A → BCB | aAb | CaDb | B
B → c | dc | e | f | g | A

A → BCB
→ aAb
→ CaDb
→ B

B → cB'
→ dcB'
→ eB'
→ fB'
→ gB'
→ aAbB'
→ CaDbB'

B' → CB
→ B'
→ ε

Squashing fragment and expression:

expression() → frag() binOp() frag() ✓
→ <LB> expr() <RB> ✓
→ <id> <LB> argList() <RB>
→ frag() ✓

frag() → <ID> |<-><ID> |<NUM>|<TRUE>|<FALS
→ expr() ✗

---

expr() → <LB> expr() <RB>
→ (<ID>|<->|<NUM>|<T>|<F>) (binOp() ex
→ <ID> <LB> argList() <RB>

LR with '(' in condition.

cond() → <~> cond() cond-prime()
→ <LB> cond() <RB> cond-prime()
→ expr() compOpp() expr() cond-prime()

expr() → <LB> expr() <RB>
→ <ID> expr_id()
→ (<-><ID>|<Num>|<T>|<F>) (binOp() expr(

⇓⇓

cond() → <~>...
→ <LB>...
→ (<LB> expr() <RB>) compOpp() expr() co
→ (<ID> expr_id()) compOpp() expr() cond'()
→ ((<~><ID>|<N>|<T>|<F>) (binOp() expr())?) com