**Analysis**

**List of left recursions and ambiguities**

These are some of the left recursions. Though some are not obvious, any production with <arithExpr> in it causes some left recursion to happen. Also there will be more left recursions when you start transforming the grammar.

<expr> ::= <arithExpr> | <relExpr>

<relExpr> ::= <arithExpr><relOp><arithExpr>

<arithExpr> ::= <arithExpr><addOp><term> | <term>

<term> ::= <term><multOp><factor> | <factor>

<factor> ::= <variable>

| <idnest>\*id(<aParams>)

| num

| (<arithExpr>)

| not<factor>

| <sign><factor>

<indice> ::= [<arithExpr>]

<aParamsTail> ::= ,<expr>

There were a lot of ambiguities, and I list some below. However, as you transform the grammar, it may cause more ambiguity somewhere else, so you have to be very careful. Also even if you left factor or remove left recursion, it does not mean that it removes the ambiguity, so you must be intelligent in how you transform your grammar.

**Description of method used to apply changes to the original grammar**

Transformed all \*notations to right recursive list generating productions

Ex:

nonTerminal\*

to

nonTerminalList -> nonTerminal nonTerminalList

Transformed all left recursive productions using the transformation technique

Ex:

A -> Aa | B

To

A -> BA’

A’ -> aA’ | Epsilon

Every time there was a common left factor, I applied the transformation technique

Ex:

A -> aB | aC

To

A -> a

ARest -> B | C

Note: You will see in my grammar things like Rest, or RestRest, this means something has been transformed.

I applied a mix of these in some order until I got a grammar that was LL1.

**Some things I changed**

**<classDecl> ::= class id {<varDecl>\*<funcDef>\*};**

This means that varDecl come first then funcDefs. We can’t mix them or put varDecls after funcDefs.

So I fixed this by merging them into one production **“varFuncDefList”.** Also since both their productions can start with type, there was an ambiguity here. So I left factored this.

**<funcbody> ::= {<varDecl>\*<statement>\*}**

This means that varDecl come first then statements. We can’t mix them or put varDecls after statements.

So I fixed this by merging them into one production **“varStatementList”.** Also since both their productions can start with id, there was an ambiguity here. So I left factored this.

**<statBlock> ::= {<statement>\*} | <statement> | EPSILON**

Has been turned into **<statBlock> ::=** {**“statementList”}**

I removed some productions because they were merged when doing left factoring.

Productions such as <funchead>, <varDecl> (I still have it in my grammar for completeness sake, but it is not being used).

Also since statement ands varDecls have a common id in them, I made a separate **“noIdStartStatement”** production. These are for statements that don’t start with id, such as for, if, get,put, return. However they can still be made from statements productions. They are just wrappers for easier parsing.

I updated the if/else production to optionally not have else blocks if not needed. In the provided grammar

**<statment> ::= if(<expr>)then<statBlock>else<statBlock>;** Made the else block be there all the time, even if not needed. So I simply made the else optional.

I split my assignStat in two. One for when the assignStarts with an id, and when it doesn’t. This uses the Rest and RestRest notation I used. This means that I had to left factor twice.

There was a lot of trouble with the **idnest** and id. There was a lot of ambiguity where it didn’t know which production to use to generate the last id. For example with <variable> it has idnest and id in it. This is an ambiguity. Even has you transform your grammar, it may not work nicely. So to solve this, I remove idnest, and introduced dotidnest. Essentially what this does, it make id be the start of the variable, and use the iddotnest to generate all the other . ids. It can generate id, id.id, id.id.id, …. Or id[id], id[id][id], id.id[id], … etc. This was very complicated and may not be the best solution, but I found it more intuitive and easier to implement and solved the ambiguity and other problems.

There was also an ambiguity with **<type>,** it wasn’t with the type itself, but when type was being used in other productions. So I did this the same way I did the assignStat. I split it into two, types that don’t start with type and type that starts with id.

There were also a lot of problems with expr, arithExpr, term, and factor. There were ambiguities with starting with id coming from variable, idnest, type. I solved this using a mix of a lot of techniques. I can’t give you an exact way how I solved it step by step.

There was also a small mistake with **indice -> [ arithExpr ]**

The problem with this is that we could have an arithExpr that evaluations to a non integer value.

So we’ll have code like **x = array[2.3].**  This is incorrect. In order to solve this problem I made a arithExprAEInt that is basically the same as arithExpr, but it ultimately evaluates to an integer value in the end.

So the production becomes **indice -> [ arithExprAEInt** **]**

So now we can code like x = array[2] or x = array[2 + id], and **x = array[2.3] is an error.**

A lot of problems and ambiguities came from the fact that a lot of things started with id. Solving all these ambiguities took a long process of a couple of days of trying different techniques and using atocc to help.

Overall, the transformed grammar might not be perfect or clean, but it works and is LL1.

**Transformed grammar**

See grammar.txt for the transformed grammar

**First/follow sets**

See first\_and\_follow\_sets.txt

**Parsing table**

See parsingtable.html in your browser to view the parsing table for this grammar

**Derivations**

See derivation.html in your browser to view the derivation for the specified source file

**Design/Implementation**

**Overall structure and reasoning on implementation**

I built a parser generator using an LL(1) table parser. You can feed a grammar file into it and it will generate the first/follow sets, the parsing table, and use the table parser’s generic algorithm to parse it. However since my lexer is hardcoded, it may not be as flexible as it should be. I wanted to do it this way because it would ultimately be less work in the end. Though a table parser is more abstract then the recursive descent parser, I think it is much easier to work with. It is easier to work with because it is shorter to write that the recursive descent parser, and much less prone to typing errors which can cause bugs. Having 30+ methods to write using a recursive descent parser, means 30+ ways of typos and subtle mistakes. Also having to change the grammar after would be a massive headache once a recursive descent parser is done. Building the parser generator allowed me to know have to hardcode the first/follow sets, and parse table. Once it is implemented right, it will work for any grammar. Also since I have some grammars to test it with, I know it is correct if it gives me the same output as I am expecting. This means once I run my grammar through it, I know it is correct. It gave me the same first/follow sets as the atocc program.

**Error recovery strategy**

I used the no panic mode using variation 3. Combined first(A) and follow (A). I simply followed the code in the slides. However during testing, I found that I was getting duplicate error outputs that came from the same place. In order to avoid outputting duplicate errors, I stored the errors and discarded new errors that were duplicates.

The errors messages will output on which line there is an error and after which token in that line that it finds the error.

The error recovery mode is extremely basic. It outputs basic errors messages and it doesn’t attempt to correct it or anything. However it will continue parsing the entire document. Though it parses the entire document, it may not find later errors due to errors at the start of the file. I tried resolving this error, but it came extremely unwieldy to solve. However as you fix the errors at the start the errors later in the file will show up. It will always find the error, and it will always finish parsing the code and will not stop when it finds an error.

**Use of tools**

**Tools/Libraries**

* I didn’t use any external tools/libs/dlls to write any code. Only used the internal libs provided with C++.
* Used atocc to help transform the grammar
* Used Bitbucket/Git for version control.
* Used STL containers such as <vector> and <unordered\_map> to hold data over simple arrays.
* Used C++11 smart pointers for easier pointer management.
* Used the unit testing framework provided inside visual studio to create unit tests.

**Techniques**

* Built parser generator using a table parser to generate all the necessary data on the spot. This avoided me having to hardcode the first/follow sets, the parse table, and avoid typos if I were to write a recursive descent parser. In the end this was much faster and easier to work with.