David Montgomery

CS4013

Project 2

August 1, 2021

Introduction:

In these projects we were asked to start to create the front end of a compiler that could be used to compile the Pascal language. This project, in particular, we created a syntax analyzer that utilized the code we wrote in project one and the grammar in LL(1) form to get a token, match this token, and output whether or not this token is expected. Every compiler has a form of grammar that the user is required to follow to compile, and pascal is no different. For this project I created a parse function, gettoken function, match function, and individual functions for all 39 terminals listed in the grammar.

Methodology:

This project began with pen and paper. We were asked to take the syntax of the pascal subset in LALR(1) form, and modify it to be in LL(1) form. Once the grammar is in this form, the syntax analyzer has to use recursive descent parsing and use the tokens that are given to us by the lexical analyzer in project 1. We were told to build a top down LL(1) parser because it is the most efficient type of parser. The first step to transition this grammar from LALR(1) to LL(1) is by eliminating all left recursion. This is because top down parsers are unable to handle left recursion. Left recursion happens in a grammar whenever there is some non-terminal variable that has a derivation where the non-terminal variable on the left side produces that same non-terminal variable on the right. This will create an infinite loop in a top down parser. There is a problem with removing left recursion if there is epsilon in the grammar, so the initial step is to remove epsilon without removing the string that the grammar generates. To complete epsilon production elimination I broke down each production rule into all of its components. If one of these included epsilon, I rewrote by mimicking the behaviour of the nullable variables. I would then rewrite the final form without the epsilon. After removing all of the epsilons in the grammar, I was able to go forward with removal of the left recursion. Again, if the non-terminal variable on the left produced the same non-terminal variable on the right we had left recursion. We got rid of this using an algorithm. The right side non-terminal variable that is produced gets changed to the knotted version of this variable. For example, if the production is A->b+A we would change this to A->b+A'. The production rule for A' would then be A'->A. This is the algorithm at its simplest form, but I went through all of the grammar and eliminated all cases of left recursion. The removal of epsilon removal and left recursion can be seen below:

1.1 program -> **program id** (idlst) ; decs sdecs cstmt.
1.2 program -> **program id** (idlst) ; sdecs cstmt.
1.3 program -> **program id** (idlst) ; decs cstmt.
1.4 program -> **program id** (idlst) ; cstmt.

1.1 program -> **program id** (idlst) ; program'

1.2 program’ -> sdecs cstmt.
1.3 program’ -> cstmt.
1.4 program’ -> decs program’’
1.5 program’’ -> sdecs cstmt.
1.6 program’’ -> cstmt.

2.1 Idlst -> **id** idlst’
2.2 Idlst’ -> , **id** idlst’

3.1 Decs -> Decs **var id** : type ;
3.2 Decs -> **var id** : type ;

4.1 Type -> stype
4.2 Type -> **array** [**num** .. **num**] **of** stype

5.1 Stype -> **integer**
5.2 Stype -> **real**

6.1 sdecs -> sdecs sdec ;
6.2 sdecs -> sdec ;
6.3 sdecs-> shead decs sdecs cstmt
6.3.1 sdecs-> shead sdecs cstmt
6.3.2 sdecs-> shead decs cstmt
6.3.3 sdecs-> shead cstmt

7.1 Sdec-> shead decs cstmt
7.2 Sdec-> shead cstmt

8.1 Shead-> **function id** args : stype ;
8.1.2 Shead-> **function id** : stype ;

9.1 args->(plist)

10.1 plist-> id : type
10.2 plist-> plist ; **id** : type

11.1 Cstmt -> **begin** ostmts **end**
11.2 Cstmt -> **begin end**

12.1 Ostmts ->slist

13.1 slist->stmt
13.2 slist->slist ; stmt

14.1 stmt->variable **assignop** expr
14.2 stmt->pstmt
14.3 stmt->cstmt
14.4 stmt->**if** expr **then** stmt **else** stmt
14.5 stmt->**while** expr **do** stmt
14.6 stmt-> **if** expr **then** expr

15.1 variable ->**id**
15.2 variable->**id** [expr]

// 16.3 pstmt->**id**
// 16.4 pstmt->**id** (elist)

17.1 elist->expr
17.2 elist->elist , expr

18.1 Expr ->sexpr
18.2 Expr->sexpr **relop** sexpr

19.1 sexpr->term
19.2 sexpr->sign term
19.3 sexpr->sexpr **addop** term

20.1 Term -> fctr
20.2 Term-> term **mulop** fctr

21.1 fctr-> **id**
21.2 fctr->**id** (elist)
21.3 fctr-> **num**
21.4 fctr-> (expr)
21.5 fctr-> **not** fctr
21.6 fctr-> id [expr]

22.1 sign-> + | -

The next step for massaging our grammar is left factoring. Left factoring occurs when you are able to factor out a portion of the production rule. For example, A->abC | abD we could factor the ab and finish with A->CA' | DA'. The A' rule would be A'->ab. I went through all of our grammar and implemented left factoring. The next step for this project was to create first and follow sets for our grammar. The first set is just the first(s) terminal variables for the production. The first of the production rule A->aBCd would be 'a'. Now it gets more complicated because it

needs to be terminal variables. For example, the first set for the production A->BcD would be the first of B. So, if B->aBD then the first of A and B would be ‘a’. I went through the grammar and found all of the firsts. Next, we were asked to find the follow sets. The follow sets are whatever follows the non-terminal variables in the grammar. For example, the follow set for A->bcd B->Acd would be ‘c’. This is because this is the terminal variable that follows this non-terminal variable in the grammar. I went through the grammar and found all of the follow sets. The left factoring and first and follow sets can be seen below:

1.1 prog -> **program id** (idlst) ; prog’	first: {**program**} follow: {$}
1.2 prog’ -> sdecs cstmt **.**	first: sdecs(sdec(shead(**function**)))
1.3 prog’ -> cstmt **.**	first: cstmt(**begin**)
1.4 prog’ -> decs prog”	first: decs(**var**) first: {**function**, **begin var**} follow:{$}
1.5 prog” -> sdecs cstmt **.**	first: sdecs(sdec(shead(**function**)))
1.6 prog” -> cstmt **.**	first: cstmt(**begin**) first: {**function**, **begin**} follow: {$}
2.1 Idlst -> **id** idlst’	first: **id** first: {**id**} follow: {) }
2.2 Idlst’ -> , **id** idlst’	first: ,
2.3 Idlst’-> ε	first: ε first: {, ε} follow: {) }
3.1 Decs -> **var id** : type ; Decs’	first: **var** {**var**} follow: {**function**, **begin**}
3.2 Decs’ -> **var id** : type ; Decs’	first: **var**
3.3 Decs’-> ε	first: ε first: {**var**, ε} follow: {**function**, **begin**}

4.1 Type -> stype	first: stype(**integer**, **real**)
4.2 Type -> **array** [**num** .. **num**] **of** stype	first: **array** first: {**integer**, **real**, **array**} follow: {; ,)}
5.1 Stype -> **integer**	first: **integer**
5.2 Stype -> **real**	first: **real** first: {**integer**,**real**} follow: {; ,)}
6.1 sdecs -> sdec ; sdecs’	first: sdec(shead(**function**)) first: {**function**} follow: {**begin**}
6.2 sdecs’ -> sdec ; sdecs’	first: sdec(shead(**function**))
6.3 sdecs’ -> ε	first: ε first: {**function**, ε} follow: {**begin**}
7.1 sdec -> shead sdec’	first: shead(**function**) first: {**function**} follow: { ; }
7.2 sdec’-> cstmt	first: cstmt(**begin**)
7.3 sdec’ -> sdecs cstmt	first: sdecs(sdec(shead(**function**)))
7.4 sdec’-> decs sdec’’	first: decs(**var**) first: {**begin**, **function**, **var** } follow: { ; }
7.5 sdec’’-> sdecs cstmt	first: sdecs(sdec(shead(**function**)))
7.6 sdec’’ -> cstmt	first: cstmt(**begin**) first: {**function**, **begin**} follow: { ; }

8.1 Shead-> **function id** stype ; shead'
first: {**function**}
follow: { **begin**, **function**, **var** }

8.2 Shead'-> args :
first: args(( )

8.3 Shead'-> :
first: :
first: { (, : }
follow: { **begin**, **function**, **var** }

9.1 args-> (plist)
first: { (}
follow: { : }

10.1 plist-> **id** : type plist'
first: **id**
first: {**id**}
follow: {) }

10.2 plist'-> ; **id** : type plist'
first: ;

10.3 plist'-> ε
first: ε
first: {; , ε}
follow: {) }

11.1 Cstmt -> **begin** cstmt'
first: {**begin**}
follow: { **.** , ; , **end**, **else**}

11.2 Cstmt' -> ostmts **end**
first: ostmts(stmt(variable(**id**), cstmt(**begin**), **while**, **if**))

11.3 Cstmt' -> **end**
first: **end**
first: {**id**, **begin**, **while**, **if**, **end** }
follow: { **.** , ; , **end**, **else**}

12.1 Ostmts ->slist
first: stmt(variable(**id**), cstmt(**begin**), **while**, **if**)
first: {**id**, **begin**, **while**, **if**}
follow: {**end**}

13.1 slist->stmt slist'
first: stmt(variable(**id**), cstmt(**begin**), **while**, **if**)
first: {**id**, **begin**, **while**, **if**}
follow: {**end**}

13.2 slist'-> ; stmt slist'	first: ;
13.3 slist' -> ε	first: ε first: {; , ε} follow: {**end**}
14.1 stmt->varisable **assignop** expr	first: variable(**id**)
14.2 stmt->cstmt	first: cstmt(**begin**)
14.3 stmt->**while** expr **do** stmt	first: **while**
14.4 stmt->**if** expr **then** stmt stmt'	first: **if** first: {**id**, **begin**, **while**, **if**} follow: { ; , **end**, **else** }
14.5 stmt'-> **else** stmt	first: **else**
14.6 stmt' -> ε	first: ε first: {**else**, ε} follow: {; , **end**, **else**}
15.1 variable ->**id** variable'	first: **id** first: {**id**} follow: {**assignop**}
15.2 variable' -> [expr]	first: [
15.3 variable' -> ε	first: ε first: { [, ε} follow: {**assignop**}
16.1 elist-> expr elist'	first: expr(sexpr(term(fctr(num, (, not, **id**))) AND sign(+,-) first: {**num** , (, **not** , **id** , + , - } follow: {) }
16.2 elist'-> , expr elist'	first: ,
16.3 elist'-> ε	first: ε first: {, ε} follow: {) }

17.1 Expr -> sexpr expr'
first: sexpr(term(fctr(num, (, not, **id**))) AND sign(+,-)
first: {**num** , (, **not** , **id** , + , - }
follow: {; , **end**, **do**, **then**,] , , ,) , **else**}

17.2 Expr' -> **relop** sexpr
first: **relop**

17.3 Expr' -> ε
first: ε
first: {**relop** , ε}
follow: {; , **end**, **do**, **then**,] , , ,), **else** }

18.1 sexpr->term sexpr'
first: term (fctr(**num**, (, **not**, **id**)

18.2 sexpr->sign term sexpr'
first: sign(+, -)
first: {**num** , (, **not** , **id** , + , - }
follow: {**relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

18.3 sexpr'-> **addop** term sexpr'
first: **addop**

18.4 sexpr'-> ε
first: ε
first: {**addop**, ε}
follow: {**relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

19.1 Term -> fctr term'
first: fctr (num, (, not, **id**)
first: {**num**, (, **not**, **id**}
follow: {**addop**, **relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

19.2 Term'-> **mulop** fctr term'
first: mulop

19.3 Term'-> ε
first: ε
first: {**mulop** , ε}
follow: {**addop**, **relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

20.1 fctr-> **num**
first: num

20.2 fctr-> (expr)
first: (

20.3 fctr-> **not** fctr
first: not

20.4 fctr-> **id** fctr'
first: **id**
first: {**num**, (, **not**, **id**}
follow: {**mulop** , **addop**, **relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

20.5 fctr' -> [expr] first: [
20.6 fctr' -> (elist) first: (
20.6 fctr' -> ε first: ε
first: {[, (, ε}
follow: {**mulop** , **addop**, **relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

21.1 sign-> + first: +
21.2 sign-> - first: -
first: {+ , -}
follow: { **num**, (, **not**, **id** }

After having the first and follow sets I was able to begin creating my parse table. A parse table has non-terminal variables on one side and terminal variables on the other. This creates a grid like table, and in the different boxes we either have a production rule or a SYNERR. A SYNERR occurs when the terminal variable is not in the first set of the non-terminal variable. If this terminal variable is included in the first set of the non-terminal variable, we put the production rule in the box that produced that variable in the first set. An example of this would be if the first of A={a} and the production rule is A->aBC. The parse table for the box created by A and a would be A->aBC. I went through and created the parse table for the massaged grammar. This parse table can be seen below:

1.1 prog -> **program id** (idlst) ; prog' first: {**program**}
follow: {$}

P[prog, program] : prog -> **program id** (idlst) ; prog'

1.2 prog' -> sdecs cstmt **.** first: sdecs(sdec(shead(**function**)))
1.3 prog' -> cstmt **.** first: cstmt(**begin**)
1.4 prog' -> decs prog'' first: decs(**var**)
first: {**function**, **begin var**}
follow:{$}

P[prog', **function**] : prog' -> sdecs cstmt **.**
P[prog', **begin**] : prog' -> cstmt **.**
P[prog', **var**] : prog' -> decs prog''

1.5 prog'' -> sdecs cstmt **.** first: sdecs(sdec(shead(**function**)))
1.6 prog'' -> cstmt **.** first: cstmt(**begin**)

first: {**function**, **begin**}
follow: {$}

P[prog'', **function**] : prog'' -> sdecs cstmt **.**
P[prog'', **begin**] : prog'' -> cstmt **.**

2.1 Idlst -> **id** idlst'

first: **id**
first: {**id**}
follow: {) }

P[Idlst, **id**] : Idlst -> **id** idlst'

2.2 Idlst' -> , **id** idlst'
2.3 Idlst'-> ε

first: ,
first: ε
first: {, ε}
follow: {) }

P[Idlst', **,**] : Idlst' -> , **id** idlst'
P[Idlst',)] : Idlst'-> ε

3.1 Decs -> **var id** : type ; Decs'

first: **var** {**var**}
follow: {**function**, **begin**}

P[Decs, **var**] : Decs -> **var id** : type ; Decs'

3.2 Decs' -> **var id** : type ; Decs'
3.3 Decs'-> ε

first: **var**
first: ε
first: {**var**, ε}
follow: {**function**, **begin**}

P[Decs', **var**] : Decs' -> **var id** : type ; Decs'
P[Decs', **function**] : Decs'-> ε
P[Decs', **begin**] : Decs'-> ε

4.1 Type -> stype

first: stype(**integer**, **real**)

4.2 Type -> **array** [**num** .. **num**] **of** stype

first: **array**
first: {**integer**, **real**, **array**}
follow: {; ,)}

P[Type, **integer**] : Type -> stype
P[Type, **real**] : Type -> stype
P[Type, **array**] : Type -> **array** [**num** .. **num**] **of** stype

5.1 Stype -> **integer**
5.2 Stype -> **real**

first: **integer**
first: **real**
first: {**integer**,**real**}
follow: {; ,)}

P[Stype, **integer**] : Stype -> **integer**
P[Stype, **real**] : Stype -> **real**

6.1 sdecs -> sdec ; sdecs'

first: sdec(shead(**function**))
first: {**function**}
follow: {**begin**}

P[sdecs, **function**] : sdecs -> sdec ; sdecs'

6.2 sdecs' -> sdec ; sdecs'
6.3 sdecs' -> ε

first: sdec(shead(**function**))
first: ε
first: {**function**, ε}
follow: {**begin**}

P[sdecs', **function**] : sdecs -> sdec ; sdecs'
P[sdecs', **begin**] : sdecs' -> ε

7.1 sdec -> shead sdec'

first: shead(**function**)

first: {**function**}
follow: { ; }

P[sdec, **function**] : sdec -> shead sdec'

7.2 sdec'-> cstmt — first: cstmt(**begin**)
7.3 sdec' -> sdecs cstmt — first: sdecs(sdec(shead(**function**)))
7.4 sdec'-> decs sdec'' — first: decs(**var**)
first: {**begin**, **function**, **var** }
follow: { ; }

P[sdec', **begin**] : sdec'-> cstmt
P[sdec', **function**] : sdec' -> sdecs cstmt
P[sdec', **var**] : sdec'-> decs sdec''

7.5 sdec''-> sdecs cstmt — first: sdecs(sdec(shead(**function**)))
7.6 sdec'' -> cstmt — first: cstmt(**begin**)
first: {**function**, **begin**}
follow: { ; }

P[sdec'', **function**] : sdec''-> sdecs cstmt
P[sdec'', **begin**] : sdec'' -> cstmt

8.1 Shead-> **function id** stype ; shead' — first: {**function**}
follow: { **begin**, **function**, **var** }

P[Shead, **function**] : Shead-> **function id** stype ; shead'

8.2 Shead'-> args : — first: args(( )
8.3 Shead'-> : — first: :
first: { (, : }
follow: { **begin**, **function**, **var** }

P[Shead', ( ] : Shead'-> args :
P[Shead', ;] : Shead'-> :

9.1 args-> (plist) — first: { (}
follow: { : }

P[args, ( ] : args-> (plist)

10.1 plist-> **id** : type plist’
first: **id**
first: {**id**}
follow: {) }

P[plist, **id**] : plist-> **id** : type plist’

10.2 plist’-> ; **id** : type plist’
first: ;

10.3 plist’-> ε
first: ε
first: {; , ε}
follow: {) }

P[plist’, **;**] : plist’-> ; **id** : type plist’
P[plist’, **)**] : plist’-> ε

11.1 Cstmt -> **begin** cstmt’
first: {**begin**}
follow: { **.** , ; , **end**, **else**}

P[Cstmt, **begin**] : Cstmt -> **begin** cstmt’

11.2 Cstmt’ -> ostmts **end**
first: ostmts(stmt(variable(**id**), cstmt(**begin**), **while**, **if**))

11.3 Cstmt’ -> **end**
first: **end**
first: {**id**, **begin**, **while**, **if**, **end** }
follow: { **.** , ; , **end**, **else**}

P[Cstmt’, **id**] : Cstmt’ -> ostmts **end**
P[Cstmt’, **begin**] : Cstmt’ -> ostmts **end**
P[Cstmt’, **while**] : Cstmt’ -> ostmts **end**
P[Cstmt’, **if**] : Cstmt’ -> ostmts **end**
P[Cstmt’, **end**] : Cstmt’ -> **end**

12.1 Ostmts ->slist
first: stmt(variable(**id**), cstmt(**begin**), **while**, **if**)

first: {**id**, **begin**, **while**, **if**}
follow: {**end**}

P[Ostmts, **id**] : Ostmts ->slist
P[Ostmts, **begin**] : Ostmts ->slist
P[Ostmts, **while**] : Ostmts ->slist
P[Ostmts, **if**] : Ostmts ->slist

13.1 slist->stmt slist'

first: stmt(variable(**id**), cstmt(**begin**), **while**, **if**)
first: {**id**, **begin**, **while**, **if**}
follow: {**end**}

P[slist, **id**] : slist->stmt slist'
P[slist, **begin**] : slist->stmt slist'
P[slist, **while**] : slist->stmt slist'
P[slist, **if**] : slist->stmt slist'

13.2 slist'-> ; stmt slist'
13.3 slist' -> ε

first: ;
first: ε
first: {; , ε}
follow: {**end**}

P[slist', ;] : slist'-> ; stmt slist'
P[slist', **end**] : slist' -> ε

14.1 stmt->varisable **assignop** expr
14.2 stmt->cstmt
14.3 stmt->**while** expr **do** stmt
14.4 stmt->**if** expr **then** stmt stmt'

first: variable(**id**)
first: cstmt(**begin**)
first: **while**
first: **if**
first: {**id**, **begin**, **while**, **if**}
follow: { ; , **end**, **else** }

P[stmt, **id**] : stmt->variable **assignop** expr
P[stmt, **begin**] : stmt->cstmt
P[stmt, **while**] : stmt->**while** expr **do** stmt
P[stmt, **if**] : stmt->**if** expr **then** stmt stmt'

14.5 stmt'-> **else** stmt
14.6 stmt' -> ε

first: **else**
first: ε
first: {**else**, ε}

follow: {; , **end**, **else**}

P[stmt', **else**] : stmt'-> **else** stmt
P[stmt', **;**] : stmt' -> ε
P[stmt', **end**] : stmt' -> ε
*P[stmt', **else**] : stmt' -> ε

15.1 variable ->**id** variable'

first: **id**
first: {**id**}
follow: {**assignop**}

P[variable, **id**] : variable ->**id** variable'

15.2 variable' -> [expr]
15.3 variable' -> ε

first: [
first: ε
first: { [, ε}
follow: {**assignop**}

P[variable', [ ] : variable' -> [expr]
P[variable', **assignop**] : variable' -> ε

16.1 elist-> expr elist'

first: expr(sexpr(term(fctr(num, (, not, **id**))) AND sign(+,-)
first: {**num** , (, **not** , **id** , + , - }
follow: {) }

P[elist, **num**] : elist-> expr elist'
P[elist, ( ] : elist-> expr elist'
P[elist, **not**] : elist-> expr elist'
P[elist, **id**] : elist-> expr elist'
P[elist, +] : elist-> expr elist'
P[elist, -] : elist-> expr elist'

16.2 elist'-> , expr elist'
16.3 elist'-> ε

first: ,
first: ε
first: {, ε}
follow: {) }

P[elist', ,] : elist'-> , expr elist'

P[elist',)] : elist'-> ε

17.1 Expr -> sexpr expr'

first: sexpr(term(fctr(num, (, not, **id**))) AND sign(+,-)
first: {**num** , (, **not** , **id** , + , - }
follow: {; , **end**, **do**, **then**,] , , ,) , **else**}

P[Expr, **num**] : Expr -> sexpr expr'
P[Expr, (] : Expr -> sexpr expr'
P[Expr, **not**] : Expr -> sexpr expr'
P[Expr, **id**] : Expr -> sexpr expr'
P[Expr, +] : Expr -> sexpr expr'
P[Expr, -] : Expr -> sexpr expr'

17.2 Expr' -> **relop** sexpr
17.3 Expr' -> ε

first: **relop**
first: ε
first: {**relop** , ε}
follow: {; , **end**, **do**, **then**,] , , ,), **else** }

P[Expr', **relop**] : Expr' -> **relop** sexpr
P[Expr', ;] : Expr' -> ε
P[Expr', **end**] : Expr' -> ε
P[Expr', **do**] : Expr' -> ε
P[Expr', **then**] : Expr' -> ε
P[Expr',]] : Expr' -> ε
P[Expr', ,] : Expr' -> ε
P[Expr',)] : Expr' -> ε
P[Expr', **else**] : Expr' -> ε

18.1 sexpr->term sexpr'
18.2 sexpr->sign term sexpr'

first: term (fctr(**num**, (, **not**, **id**)
first: sign(+, -)
first: {**num** , (, **not** , **id** , + , - }
follow: {**relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

P[sexpr, **num**] : sexpr->term sexpr'
P[sexpr, ( ] : sexpr->term sexpr'
P[sexpr, **not**] : sexpr->term sexpr'
P[sexpr, **id**] : sexpr->term sexpr'
P[sexpr, +] : sexpr->term sexpr'
P[sexpr, -] : sexpr->term sexpr'

18.3 sexpr'-> **addop** term sexpr' first: **addop**
18.4 sexpr'-> ε first: ε
first: {**addop**, ε}
follow: {**relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

P[sexpr', **addop**] : sexpr'-> **addop** term sexpr'
P[sexpr', **relop**] : sexpr'-> ε
P[sexpr', ;] : sexpr'-> ε
P[sexpr', **end**] : sexpr'-> ε
P[sexpr', **do**] : sexpr'-> ε
P[sexpr', **then**] : sexpr'-> ε
P[sexpr',]] : sexpr'-> ε
P[sexpr', ,] : sexpr'-> ε
P[sexpr',)] : sexpr'-> ε
P[sexpr', **else**] : sexpr'-> ε

19.1 Term -> fctr term' first: fctr (num, (, not, **id**)
first: {**num**, (, **not**, **id**}
follow: {**addop**, **relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

P[Term, **num**] : Term -> fctr term'
P[Term, ( ] : Term -> fctr term'
P[Term, **not**] : Term -> fctr term'
P[Term, **id**] : Term -> fctr term'

19.2 Term'-> **mulop** fctr term' first: mulop
19.3 Term'-> ε first: ε
first: {**mulop** , ε}
follow: {**addop**, **relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

P[Term', **mulop**] : Term'-> **mulop** fctr term'
P[Term', **addop**] : Term'-> ε
P[Term', **relop**] : Term'-> ε

P[Term', ;] : Term'-> ε
P[Term', **end**] : Term'-> ε
P[Term', **do**] : Term'-> ε
P[Term', **then**] : Term'-> ε
P[Term',]] : Term'-> ε
P[Term', ,] : Term'-> ε
P[Term',)] : Term'-> ε
P[Term', **else**] : Term'-> ε

20.1 fctr-> **num** first: num
20.2 fctr-> (expr) first: (
20.3 fctr-> **not** fctr first: not
20.4 fctr-> **id** fctr' first: **id**
first: {**num**, (, **not**, **id**}
follow: {**mulop** , **addop**, **relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

P[fctr, **num**] : fctr-> **num**
P[fctr, ( ] : fctr-> (expr)
P[fctr, **not**] : fctr-> **not** fctr
P[fctr, **id**] : fctr-> **id** fctr'

20.5 fctr' -> [expr] first: [
20.6 fctr' -> (elist) first: (
20.6 fctr' -> ε first: ε
first: {[, (, ε}
follow: {**mulop** , **addop**, **relop** , ; , **end**, **do**, **then**,] , , ,), **else**}

P[fctr, [ ] : fctr' -> [expr]
P[fctr, ( ] : fctr' -> (elist)
P[fctr, **mulop**] : fctr' -> ε
P[fctr, **addop**] : fctr' -> ε
P[fctr, **relop**] : fctr' -> ε
P[fctr, ;] : fctr' -> ε
P[fctr, **end**] : fctr' -> ε
P[fctr, **do**] : fctr' -> ε
P[fctr, **then**] : fctr' -> ε
P[fctr,]] : fctr' -> ε
P[fctr, ,] : fctr' -> ε
P[fctr,)] : fctr' -> ε
P[fctr, **else**] : fctr' -> ε

21.1 sign-> +
21.2 sign-> -

first: +
first: -
first: {+ , -}
follow: { **num**, (, **not**, **id** }

P[sign, +] : sign-> +
P[sign, -] : sign-> -

This is the completion of the written part for project two. I will use this parse table to create the non-terminal variable functions using switch statements.

Implementation:

After concluding the grammar transformation to LL(1), I began to create the functions necessary to implement this grammar into code. I began this project by modifying the code that I had written for project 1. I used a while loop in project 1 to parse every token line by line. I was unable to use this while loop for project 2 because I needed to pass these tokens to the syntax analyzer token by token and not line by line. Instead of the initial while loop in project 1, I created a parse function that is called at the end of the main function. This parse function does not return anything, but will set a global token equal to gettoken.

tok = gettoken();

This gettoken function is just a modified version of my main function in project 1. The purpose of this gettoken function is to produce, print, and return the first token of the input file. The biggest difference between this function and the main function in project one is I am returning the tokens one by one. In the previous project I only printed the tokens, so I was able to utilize a while loop to continuously print the tokens until the file was empty. Instead of using a while loop I would check if the value inside the buffer, or the ptr, is equal to '\0'. This if statement will check for the end of the file. The if statement can be seen here:

```
if(buffer[b] != '\0')
```

If the ptr is the end of the file, it will move to the else statement that will return the end of file token. That will look like this:

```
token *eof = malloc(sizeof(struct token))
strcpy(eof->lex, "EOF")
strcpy(eof->tokenname, "ENDFILE")
eof->types = ENDFILE
strcpy(eof>attname, "NIL")
eof>types.att = NIL
fprintf(tokenoutput, "%d.       %s %s %d %s %d \n", i, eof->lex, eof->tokenname, eof->type,
eof->attname, eof->types.att
return eof
```

If the ptr is not equal to '\0' it will print the first line of the input file to the listing file. After printing, the gettoken function will use the code that I implemented in project one to get the first token in the input file. I print this token and its attributes to the token file. Lastly, I check if the returned token is a LEXERR. If it is a LEXERR, I enter an if statement that will print the particular LEXERR that the token triggers. The code for the last part is listed below:

```
if(tokenGlob->type == LEXERR)
```

Once we get the token, we will call the prog function inside parse. The prog function will check if the token is the expected token that is required to begin the coding process in pascal. For pascal this is the reserved word "program". I will talk in depth about the function prog because all of the other terminal functions act in the same manner. I begin this function with a switch statement. This switch statement stores the type of the token that was returned earlier.

```
switch(tok->type)
```

The first, and only, case for this function is the type PROG. This is the type for the reserved word “program”. If the type is not PROG, it will enter the default case. This default case will print a SYNERR message to the listing file. The SYNERR will read “SYNERR: Expecting program. Received _____ “ This blank is whatever the token type of the returned token was. It will then loop through the tokens until the token type is the eof token. This is specific for the prog function because pascal code needs to begin with the key word “program”. Without this keyword the code will not compile. You can see this here:

```
default:
        fprintf(listing, “SYNERR: Expected program. Received %s\n”, tok->tokenname)
        while(tok->type!= ENDFILE )
        {
                tok = gettoken();
        }
```

If the case statement is true, we will match PROG.

```
match(PROG)
```

In the match function, we take in some token type. This type is an int because we defined it in our header file for project 1. This token type is what is expected of the input file because of the grammar that needs to be followed for the pascal language. Match will then use an if statement to check if the token type of the returned token is equal to the token type that is expected. It will check if the token type is the EOF token. If it is the EOF token, this means that we have parsed all of the tokens and we break out of the match function.

```
void match(int t) {

if(tok->type == t && tok->type == ENDFILE)
{
        printf(“Parse complete.\n”)
        return
}
```

If the returned token type is equal to the expected, but the type is not the EOF token we get the next token by setting the global token variable equal to the gettoken function.

```
else if(tok->type==t && tok->type!= ENDFILE)
{
        tok = gettoken()
```

```
}
```

Lastly, we check if the returned token type is not equal to the expected token. If this happens, we print what we were expecting and set the global token variable equal to the gettoken function.

```
else if(tok->type != t)
{
        printf("%s %d", "SYNERR: EXPECTING ", t)
        tok = gettoken()
}
```

This is the end of the match function. If the token type was not the EOF token, the next expected type is ID. The same process that happened for the PROG type happens to the ID type. This process continues throughout the terminal function. Listed below are all of the match functions inside the PROG function.

```
case PROG:
        match(PROG)
        match(ID)
        match(OP)
        idLst()
        match(CP)
        match(SEMICOLON)
        progTail()
        break
```

As you can see, not every function being called is a match function. The idLst function, like all of the terminal functions, has a switch statement. The case for this switch statement is ID. Each terminal function will call, depending on the grammar, what is supposed to be matched and other terminal functions. The way this process ends is if one of the cases is null terminated or we reach the end of a case. In either case, we break from this terminal function. Take the idLstTail for example, there are two cases and the second, CP (closed parenthesis), is a null terminator. This can be seen below:

```
case CP:
        break;
```

Once the input file is empty, we will break from the prog terminal function. The last step of this project 2 is to match the endfile. This means that when we get an EOF token we need to match it and make sure that this token matches what is expected. If the returned eof token matches what is expected, I print out “Parse Complete” and return from the match function.

This completes project 2.

Discussion and Conclusions

Overall, this project was a bit easier than the first project, however, it was still very difficult. The most difficult part was the handwritten grammar transformation and not the actual implementation of code. Finding the first and follow sets and creating the parse table for my grammar was a long and meticulous process.

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (1986). Compilers: Principles, techniques, & tools.

Appendix 1

Input file 1

```
program test (input, output);
  var a : integer;
  var b : real;
  var c : array [1..2] integer;
    begin
      a:= fun1(x, e, c, b);
      x:= fun3(c[1], e);
      e := e + 4.44;
      a:= (a mod y) div x;
        while ((a >= 4) and ((b <= e)
          or (not (a = c[a])))) do
      begin
        a:= c[a] + 1
      end
```

end.

Token file 1

1. program PROG 11 NIL 50
1. test ID 12 ADDRESS 887095872
1. (OP 13 NIL 50
1. input ID 12 ADDRESS 2044724672
1. , COMMA 14 NIL 50
1. output ID 12 ADDRESS 2044725328
1.) CP 15 NIL 50
1. ; SEMICOLON 16 NIL 50
2. var VAR 17 NIL 50
2. a ID 12 ADDRESS 1262498544
2. : COLON 18 NIL 50
2. integer INTEGER 45 NIL 50
2. ; SEMICOLON 16 NIL 50
3. var VAR 17 NIL 50
3. b ID 12 ADDRESS 1648363152
3. : COLON 18 NIL 50
3. real REAL 24 NIL 50
3. ; SEMICOLON 16 NIL 50
4. var VAR 17 NIL 50
4. c ID 12 ADDRESS 1648363168
4. : COLON 18 NIL 50
4. array ARRAY 43 NIL 50
4. [OB 19 NIL 50
4. 1 NUM 55 INT 23
4. .. DOTDOT 21 NIL 50
4. 2 NUM 55 INT 23
4.] CB 22 NIL 50
4. integer INTEGER 45 NIL 50
4. ; SEMICOLON 16 NIL 50
5. begin BEGIN 27 NIL 50
6. a ID 12 ADDRESS 1262498544
6. := ASSIGNOP 29 NIL 50
6. fun1 ID 12 ADDRESS 1648385136
6. (OP 13 NIL 50
6. x ID 12 ADDRESS 1648385920
6. , COMMA 14 NIL 50
6. e ID 12 ADDRESS 1648415392

6. , COMMA 14 NIL 50
6. c ID 12 ADDRESS 1648363168
6. , COMMA 14 NIL 50
6. b ID 12 ADDRESS 1648363152
6.) CP 15 NIL 50
6. ; SEMICOLON 16 NIL 50
7. x ID 12 ADDRESS 1648385920
7. := ASSIGNOP 29 NIL 50
7. fun3 ID 12 ADDRESS 1648417072
7. (OP 13 NIL 50
7. c ID 12 ADDRESS 1648363168
7. [OB 19 NIL 50
7. 1 NUM 55 INT 23
7.] CB 22 NIL 50
7. , COMMA 14 NIL 50
7. e ID 12 ADDRESS 1648415392
7.) CP 15 NIL 50
7. ; SEMICOLON 16 NIL 50
8. e ID 12 ADDRESS 1648415392
8. := ASSIGNOP 29 NIL 50
8. e ID 12 ADDRESS 1648415392
8. + ADDOP 56 PLUS 48
8. 4.44 NUM 55 REALNUM 63
8. ; SEMICOLON 16 NIL 50
9. a ID 12 ADDRESS 1262498544
9. := ASSIGNOP 29 NIL 50
9. (OP 13 NIL 50
9. a ID 12 ADDRESS 1262498544
9. mod MULOP 57 MOD 62
9. y ID 12 ADDRESS 1648387344
9.) CP 15 NIL 50
9. div MULOP 57 DIV 60
9. x ID 12 ADDRESS 1648385920
9. ; SEMICOLON 16 NIL 50
10. while WHILE 33 NIL 50
10. (OP 13 NIL 50
10. (OP 13 NIL 50
10. a ID 12 ADDRESS 1262498544
10. >= RELOP 35 GE 40
10. 4 NUM 55 INT 23

10.) CP 15 NIL 50
10. and MULOP 57 AND 61
10. (OP 13 NIL 50
10. (OP 13 NIL 50
10. b ID 12 ADDRESS 1648363152
10. <= RELOP 35 LE 39
10. e ID 12 ADDRESS 1648415392
10.) CP 15 NIL 50
11. or ADDOP 56 OR 58
11. (OP 13 NIL 50
11. not NOT 42 NIL 50
11. (OP 13 NIL 50
11. a ID 12 ADDRESS 1262498544
11. = RELOP 35 EQ 36
11. c ID 12 ADDRESS 1648363168
11. [OB 19 NIL 50
11. a ID 12 ADDRESS 1262498544
11.] CB 22 NIL 50
11.) CP 15 NIL 50
11.) CP 15 NIL 50
11.) CP 15 NIL 50
11.) CP 15 NIL 50
11. do DO 34 NIL 50
12. begin BEGIN 27 NIL 50
13. a ID 12 ADDRESS 1262498544
13. := ASSIGNOP 29 NIL 50
13. c ID 12 ADDRESS 1648363168
13. [OB 19 NIL 50
13. a ID 12 ADDRESS 1262498544
13.] CB 22 NIL 50
13. + ADDOP 56 PLUS 48
13. 1 NUM 55 INT 23
14. end END 28 NIL 50
15. end END 28 NIL 50
15. . DOT 20 NIL 50
17. EOF ENDFILE 49 NIL 50

Listing File 1

1. program test (input, output);

```
2.      var a : integer;
3.      var b : real;
4.      var c : array [1..2] integer;
5.        begin
6.          a:= fun1(x, e, c, b);
7.          x:= fun3(c[1], e);
8.          e := e + 4.44;
9.          a:= (a mod y) div x;
10.             while ((a >= 4) and ((b <= e)
11.                 or (not (a = c[a])))) do
12.           begin
13.             a:= c[a] + 1
14.           end
15.       end.
```

16.

Input file 2

```
program test (input, output);
  var a : integer;
  var b : real;
  var c : array [1..2] ty integer;
    begin
      a:= (err here)fun1(x, e, c, b);
      x:= fun3(c[1], e);
      e := e + 4.44473745;
      #
      a:= (a mod y) ( div x;
      while ((a >= 4) and ((b <= e)
                  or (not (a = c[a])))) do
       begin
         a:= c[a] ()+ 1
end
end.
```

Token file 2

```
1.    program PROG 11 NIL 50
1.    test ID 12 ADDRESS 887095872
1.    ( OP 13 NIL 50
1.    input ID 12 ADDRESS 2044724672
1.    , COMMA 14 NIL 50
1.    output ID 12 ADDRESS 2044725328
1.    ) CP 15 NIL 50
1.    ; SEMICOLON 16 NIL 50
2.    var VAR 17 NIL 50
2.    a ID 12 ADDRESS 1262498544
2.    : COLON 18 NIL 50
2.    integer INTEGER 45 NIL 50
2.    ; SEMICOLON 16 NIL 50
3.    var VAR 17 NIL 50
3.    b ID 12 ADDRESS 1648363152
3.    : COLON 18 NIL 50
3.    real REAL 24 NIL 50
```

3. ; SEMICOLON 16 NIL 50
4. var VAR 17 NIL 50
4. c ID 12 ADDRESS 1648363168
4. : COLON 18 NIL 50
4. array ARRAY 43 NIL 50
4. [OB 19 NIL 50
4. 1 NUM 55 INT 23
4. .. DOTDOT 21 NIL 50
4. 2 NUM 55 INT 23
4.] CB 22 NIL 50
4. ty ID 12 ADDRESS -1504693504
4. integer INTEGER 45 NIL 50
4. ; SEMICOLON 16 NIL 50
5. begin BEGIN 27 NIL 50
6. a ID 12 ADDRESS 1262498544
6. := ASSIGNOP 29 NIL 50
6. (OP 13 NIL 50
6. err ID 12 ADDRESS 1044383376
6. here ID 12 ADDRESS 1044383392
6.) CP 15 NIL 50
6. fun1 ID 12 ADDRESS 1648385136
6. (OP 13 NIL 50
6. x ID 12 ADDRESS 1648385920
6. , COMMA 14 NIL 50
6. e ID 12 ADDRESS 1648415392
6. , COMMA 14 NIL 50
6. c ID 12 ADDRESS 1648363168
6. , COMMA 14 NIL 50
6. b ID 12 ADDRESS 1648363152
6.) CP 15 NIL 50
6. ; SEMICOLON 16 NIL 50
7. x ID 12 ADDRESS 1648385920
7. := ASSIGNOP 29 NIL 50
7. fun3 ID 12 ADDRESS 1648417072
7. (OP 13 NIL 50
7. c ID 12 ADDRESS 1648363168
7. [OB 19 NIL 50
7. 1 NUM 55 INT 23
7.] CB 22 NIL 50
7. , COMMA 14 NIL 50

7. e ID 12 ADDRESS 1648415392
7.) CP 15 NIL 50
7. ; SEMICOLON 16 NIL 50
8. e ID 12 ADDRESS 1648415392
8. := ASSIGNOP 29 NIL 50
8. e ID 12 ADDRESS 1648415392
8. + ADDOP 56 PLUS 48
8. 4.44473745 LEXERR 999 ERR8 908
8. ; SEMICOLON 16 NIL 50
9. # LEXERR 999 ERR1 901
10. a ID 12 ADDRESS 1262498544
10. := ASSIGNOP 29 NIL 50
10. (OP 13 NIL 50
10. a ID 12 ADDRESS 1262498544
10. mod MULOP 57 MOD 62
10. y ID 12 ADDRESS 1648387344
10.) CP 15 NIL 50
10. (OP 13 NIL 50
10. div MULOP 57 DIV 60
10. x ID 12 ADDRESS 1648385920
10. ; SEMICOLON 16 NIL 50
11. while WHILE 33 NIL 50
11. (OP 13 NIL 50
11. (OP 13 NIL 50
11. a ID 12 ADDRESS 1262498544
11. >= RELOP 35 GE 40
11. 4 NUM 55 INT 23
11.) CP 15 NIL 50
11. and MULOP 57 AND 61
11. (OP 13 NIL 50
11. (OP 13 NIL 50
11. b ID 12 ADDRESS 1648363152
11. <= RELOP 35 LE 39
11. e ID 12 ADDRESS 1648415392
11.) CP 15 NIL 50
12. or ADDOP 56 OR 58
12. (OP 13 NIL 50
12. not NOT 42 NIL 50
12. (OP 13 NIL 50
12. a ID 12 ADDRESS 1262498544

12. = RELOP 35 EQ 36
12. c ID 12 ADDRESS 1648363168
12. [OB 19 NIL 50
12. a ID 12 ADDRESS 1262498544
12.] CB 22 NIL 50
12.) CP 15 NIL 50
12.) CP 15 NIL 50
12.) CP 15 NIL 50
12.) CP 15 NIL 50
12. do DO 34 NIL 50
13. begin BEGIN 27 NIL 50
14. a ID 12 ADDRESS 1262498544
14. := ASSIGNOP 29 NIL 50
14. c ID 12 ADDRESS 1648363168
14. [OB 19 NIL 50
14. a ID 12 ADDRESS 1262498544
14.] CB 22 NIL 50
14. (OP 13 NIL 50
14.) CP 15 NIL 50
14. + ADDOP 56 PLUS 48
14. 1 NUM 55 INT 23
15. end END 28 NIL 50
16. end END 28 NIL 50
16. . DOT 20 NIL 50
17. EOF ENDFILE 49 NIL 50

Listing file 2

1. program test (input, output);

2. var a : integer;

3. var b : real;

4. var c : array [1..2] ty integer;
SYNERR: Expecting INTEGER or Real. Received ID

5. begin

6. a:= (err here)fun1(x, e, c, b);
SYNERR: Expecting mulop, addop, relop, ; , end, do, then,] , , ,) , or else. Received ID

```
SYNERR: Expecting mulop, addop, relop, ; ,  end, do, then, ] , , , ) , or else. Received ID
SYNERR: Expecting ; or end. Received COMMA

7.          x:= fun3(c[1], e);

8.          e := e + 4.44473745;

LEXERR: Number after decimal point is greater than 5 characters     4.44473745
9.          #

LEXERR: Unknown symbol:     #
10.          a:= (a mod y) ( div x;

11.          while ((a >= 4) and ((b <= e)

12.                          or (not (a = c[a])))) do

13.            begin

14.              a:= c[a] ()+ 1

15.    end

16.    end.
```

Input file 3

```
program test (input, output);
  var a : integer;
  var b : real;
  var c : array [1..2] ty integer;
    begin
      a:= (err here)fun1(x, e, c, b);
      x:= fun3(c[1], e);
      e := e + 4.44;
      a:= (a mod y) ( div x;
      while ((a >= 4) and ((b <= e)
                  or (not (a = c[a])))) do
        begin
          a:= c[a] ()+ 1
end
```

end.

Listing file 3

```
1.    program test (input, output);

2.      var a : integer;

3.      var b : real;

4.      var c : array [1..2] ty integer;
 SYNERR: Expecting INTEGER or Real. Received ID

5.        begin

6.          a:= (err here)fun1(x, e, c, b);
 SYNERR: Expecting mulop, addop, relop, ; ,  end, do, then, ] , , , ) , or else. Received ID
SYNERR: Expecting mulop, addop, relop, ; ,  end, do, then, ] , , , ) , or else. Received ID
SYNERR: Expecting ; or end. Received COMMA

7.          x:= fun3(c[1], e);

8.          e := e + 4.44;

9.          a:= (a mod y) ( div x;

10.          while ((a >= 4) and ((b <= e)

11.                        or (not (a = c[a])))) do

12.            begin

13.              a:= c[a] ()+ 1

14.    end

15.    end.
```

Appendix 2: