David Montgomery

CS4013

Project 3&4

Dec 4, 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. For project 3, in particular, we decorated our grammar to give the language the ability to do type checking and scope checking. This was done by creating a semantic analyzer that used the lex tokens developed in the first project, a decorated parse tree, and multiple error handling functions. We were asked to follow the standard scope checking rules for pascal and complete type checking. For project 4, we were asked to compute memory addresses for local variables in the scope. The memory addresses were dependent on the variable type and size.

Methodology:

This project began with pen and paper. I used the completed massaged grammar that we developed in our project 2 (listed below), and began decorating the functions to include inherited and synthesized attributes. To do this, I drew out many different parse trees and decorated them. In these drawings, I included how the information would flow and which function would need synthesized/inherited attributes. This was the longest process of the project. It took me weeks to get a good understanding of how to complete these drawings/decorate the grammar. Once I finished decorating a few of the trees, I began to edit the project two code to include these decorated additions. This was a long and tedious process, but after a long period I was able to complete this portion of the project. Now that the grammar included synthesized/inherited attributes, I was able to begin type checking. Statements, assignments, relop, addop, and mulop all needed to use inherited attributes to check if the operation was allowed. For example, an integer should not be allowed to be added to a real number. So, we kept track of variables and used the decorated parse tree and newly written error checking functions to make sure that the types being operated on did not throw any errors. I wrote a gettype function that would return the type of the variable being called within the statement and store its type in a variable. I would then move onto the next element in the input file and match it. Depending on what this matched operation was, I would then store the variable after and pass both the variable before and after the operation into a type compatibility checking function. This function would either return "okay", the return type of the operation, or an error. After completing type checking, the next step was to complete scope checking. In my opinion, this was a little bit easier for me to understand and work with. The scope checking consisted of a double linked list that stored green and blue nodes. The green nodes were function names and the blues nodes were variable names. The doubly linked list stored the color of the node, name, and type. This information was used to check the name of each local variable to make sure that the same name had not been used in the scope. It also used the name of the green nodes to make sure that the function names were not reused. After completing this data structure, I moved onto implementing it into my parse functions. This was the end of project 3, and I began the process of integrating the memory addresses for these local blue nodes. This was extremely easy and only took a couple hours. When a new function is declared, the function offset is reset to 0 and each newly added local variable increases this offset. At this point, my project 3 and 4 were basically completed and ready for a checkoff.

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’	first: ,
2.3 Idlst’-> ε	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’	first: **var**
3.3 Decs’-> ε	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**	first: **integer**
5.2 Stype -> **real**	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’	first: sdec(shead(**function**))
6.3 sdecs’ -> ε	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' first: ;
13.3 slist' -> ε first: ε
first: {; , ε}

	follow: {**end**}

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

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** }

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	first: **else**
14.6 stmt’ -> ε	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]	first: [
15.3 variable’ -> ε	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’	first: term (fctr(**num**, (, **not**, **id**)
18.2 sexpr->sign term sexpr’	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-> +	first: +
21.2 sign-> -	first: -
	first: {+ , -}
	follow: { **num**, (, **not**, **id** }

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

Implementation:

After finishing the handwritten drawings and grammar decorations, the first step that I completed was modifying the parse functions. I am not going to walk through every parse function, but I will explain a few of them. The first parse function that I will explain is the prog function. This function did not return anything, and did not need any synthesized or inherited attributes. However, we declared a green node so we needed to add it to the linked list.

```
void prog()
{
    switch (tok->type)
    {
    case PROG:
        match(PROG);
        if (tok->type == ID)
        {
            createGreenNode(tok->lex, PGMNAME);
        }
        match(ID);
        match(OP);
        idLst();
        match(CP);
        match(SEMICOLON);
        progTail();
        break;
    default:
        fprintf(listing, "SYNERR: Expecting program. Received %s\n", tok->tokenname);
        while (tok->type != ENDFILE)
        {
            tok = gettoken();
        }
    }
}
```

I start by matching the program reserved word, but instead of just matching the ID, I add this ID to the linked list. This begins the doubly linked list. The creategreennode function is a complicated function that takes in the lexName and the type. I have a nodeList that will add the first green node to it.

```
    if (nodeList == NULL)
    {


        strcpy(newNode->lex, lexName);
        newNode->type = type;

        newNode->colorNode = GREENNODE;

        newNode->prev = NULL;
        nodeList = newNode;
        eye = nodeList;
        strcpy(lastGreenNode->lex, newNode->lex);
        return;
    }
```

If this is not the first node being added to the list, I have created a loop that will check the nodeList and iterate through it until the next node is not NULL. It will also check if the type is equal to PGMNAME, which is the first node in the list. It will iterate until it reached

PGMNAME to make sure that no name is repeating. I then use STRNCMP and compare the lex name with all the names in the nodelist.

```
if (nodeList->next != NULL)
    {

        while (t != PGMNAME)
        {

            if (strncmp(lexName, last->lex, asdf) != 0)
```

If the ID name entered does not equal any of the names in the NodeList, then it will add it to the list.

```
                    newNode = nodeList;
                    prevNode = newNode;
                    newNode->nextGreen = malloc(sizeof(token));
                    newNode = newNode->nextGreen;
                    newNode->next = NULL;
                    //newNode->nextGreen = NULL;

                    strcpy(newNode->lex, lexName);
                    newNode->type = type;

                    newNode->colorNode = GREENNODE;
                    nodeList = newNode;
                    newNode->next = NULL;
                    newNode->prev = prevNode;
```

If the function name is the same as another, or the variable name is the same as another in its scope, we return SEMERR and print the error.

```
            else
            {
                fprintf(listing, "Duplicate variable/function name in scope");
```

After declaring the program function we need to create memory addresses for the local variables and add the ids to the doubly linked list. To do this, we have a decs function that matches the var reserved word, and stores the name of the variable into a variable. We retrieve the type of the variable and print the memory address that is set in the type function.

```
if (type_t != ERR && tokType != LEXERR)
        {
            fprintf(address, "%s\tloc%d\n", lex2, offset);
            offset = offset + offset2;
            offset2 = 0;
        }
```

The type function is a key function that adds the variable to the doubly linked list and also does some array type checking. Listed below is the case when the variable is an integer. We need to set the offset to 4 and add the variable to the linked list. This createBlueNode copies the same format as the createGreenNode above.

```
case INTEGER: //check
        type_t = stype();
        offset2 = 4;
        if (t == 1)
        {
            createBlueNode(lex2, INTEGER);
        }
        else if (t == 2)
        {
            createBlueNode(lex2, FUNPINT);
        }
        return type_t;
        break;
```

The type function also does some type checking for arrays. It needs to make sure that the array is written in the correct format. This array error checking function is listed below.

```
int arrayFun1(int num1, int num2, int num1_t, int num2_t)
{
    int retType;
    int diff = num2 - num1;
    if (num1 == ERRSTAR || num2 == ERRSTAR || num1_t == ERRSTAR || num2_t == ERRSTAR)
    {
        retType = ERR;
        return retType;
    }
    else if (num1_t == INTNUM && num2_t == INTNUM && diff > 0)
    {
        retType = OK;
        return retType;
    }
    else if (diff < 0)
    {
        retType = ERR;
        fprintf(listing, "SEMERR: Array declaration should list smaller number before
bigger\n");
        return retType;
    }
    else if (num1 != INTNUM || num2 != INTNUM)
    {
        retType = ERR;
        fprintf(listing, "SEMERR: Incorrect array declaration, needs to include
integers within brackets\n");
        return retType;
    }
    else
    {
        retType = ERR;
        fprintf(listing, "SEMERR: Incorrect array declaration\n");
        return retType;
    }
}
```

After declarations, we need to make sure that the offset is set equal to 0 for the next function. This is done in the sHead function which you can see below.

```
case FUNCTION:
        match(FUNCTION);
        offset = 0;
```

The next part of project 3 was the most difficult part of any of the projects, in my opinion. This is when we start using synthesized and inherited attributes to type check. One of the most important areas to type check is statements. This is where we make sure that whatever statement we are completing has the same types on both sides. Listed below is the statement checking function that makes sure that assignop, while, and if statements all follow the expected input.

```
ase ID:
        var_t = variable();
        match(ASSIGNOP);
        expr_t = expr();
        retType = stmtErr(var_t, expr_t);
        return retType;
        break;
    case BEGIN:
        cstmt();
        stmt_t = OK;
        return stmt_t;
        break;
    case WHILE:
        match(WHILE);
        expr_t = expr();
        whileErr_t = whileErr(expr_t);
        match(DO);
        stmt();
        return whileErr_t;
        break;
    case IF:
        match(IF);
        expr_t = expr();
        ifErr_t = ifErr(expr_t);
        match(THEN);
        stmt();
        stmtTail();
        return ifErr_t;
        break;
```

Listed below is the ifERr function. “T” is the type that is within the if statement. So, we need to make sure this type is not equal to ERR or anything else except BOOL. If it is, we return a SEMERR message and return ERR. Most all the other statement checking happens in the same exact way, so I will not go into detail on each of them.

```
    int retType;
    if (t == ERRSTAR || t == ERR)
    {
        retType = ERR;
        return retType;
    }
    else if (t == BOOL)
    {
        retType = OK;
        return retType;
    }
    else
    {
        retType = ERR;
        fprintf(listing, "SEMRR: If statment must compare BOOL\n");
        return retType;
    }
}
```

The last thing I will touch on is how I completed function calls. The first step is to make sure that the function type is the same as the variable you are assigning it to. So, this is done through statement type checking. If this does not result in an error then we need to check and make sure that the parameters inside the function call match up with the parameters of the function. This is what the eList function does.

```
    case ID:
        elistOutput = expr();
        if (elistOutput == REAL)
        {
            elistOutput = FUNPREAL;
        }
        else if (elistOutput == INTEGER)
        {
            elistOutput = FUNPINT;
        }
        else if (elistOutput == AREAL)
        {
            elistOutput = FUNPAREAL;
            //elistOutput = FUNPREAL;
        }
        else if (elistOutput == AINT)
        {
            elistOutput = FUNPAINT;
            //elistOutput = FUNPINT;
        }
        else if (elistOutput == REALNUM)
        {
            elistOutput = FUNPREAL;
        }
        else if (elistOutput == INTNUM)
        {
            elistOutput = FUNPINT;
        }
        else if(elistOutput == ERRSTAR) {

            while (tok->type != ENDFILE && tok->type != CP)
        {
            tok = gettoken();
        }

            return ERRSTAR; //change to get SEMERR
            //break;
        }
        eListTail_t = elistTail();
        return eListTail_t;
        break;
```

This is just part of the eList function. However, this is the most important part of it. We need to get the type of variable that is listed within the expression list. To do this, we write a getType function. That function is listed below.

```
int getType(token *gettok)
{
    int type;
    token *typeList;
    //typeList = malloc(sizeof(token));
    typeList = nodeList;
    //isitEmpty(typeList);
    if (!typeList)
    {
        return ERR;
    }
    while ((strcmp(gettok->lex, typeList->lex) != 0) && (typeList->type != PGMNAME))
    {
        typeList = typeList->prev;
    }

    if (strcmp(gettok->lex, typeList->lex) == 0)
    {
        type = typeList->type;
        if (type == 9)
        {
            type = typeList->resType;
            func = 1;
        }
        return type;
    }
    else
    {
        fprintf(listing, "SEMERR: Variable not declared\n");
        return SEMERR;
    }
}
```

This function will look at the typeList which I have equal to the doubly linked list. It will check its local scope first and try to find the variable, but if it does not find it in the local scope it will keep moving up the linked list till PGMNAME. If it reaches PGMNAME it means it is not declared or can not be read in the scope. After each new expression, we concat them to a list that we compare against the expected list as can be seen below.

```
int checkParam(int receivedParam, char p[10])
{
    //int i = 0;
    int retType;

    if (receivedParam == ERRSTAR)
    {
        retType = ERRSTAR;
        return retType;
    }
    if (receivedParam == atoi(&p[parmCheck]))
    {
        retType = OK;
        return retType;
    }
    else
    {

        retType = ERR;
        fprintf(listing, "SEMERR: Actual function parameters do not match formal 
parameters\n");
        return retType;
    }
}
```

The final piece of code that I will discuss is a function that has synthesized and inherited attributes as can be seen below.

```
int sexprTail(int sexprTail_i)
{
    int sexprTail_type;
    int sexpr_type2;
    int addop_type;
    int t_type;
    switch (tok->type)
    {
    case ADDOP:
        //addop_type = tok->types.att;
        match(ADDOP);
        t_type = term();
        sexpr_type2 = addopErr(sexprTail_i, t_type);
        sexprTail_type = sexprTail(sexpr_type2);
        return sexprTail_type;
        break;
    case RELOP:
        sexprTail_type = sexprTail_i;
        return sexprTail_type;
        break;
    case SEMICOLON:
        sexprTail_type = sexprTail_i;
        return sexprTail_type;
        break;
    case END:
        sexprTail_type = sexprTail_i;
        return sexprTail_type;
        break;
    case DO:
        sexprTail_type = sexprTail_i;
        return sexprTail_type;
        break;
    case THEN:
        sexprTail_type = sexprTail_i;
        return sexprTail_type;
        break;
    case CB:
        sexprTail_type = sexprTail_i;
        return sexprTail_type;
        break;
    case COMMA:
        sexprTail_type = sexprTail_i;
        return sexprTail_type;
        break;
    case CP:
        sexprTail_type = sexprTail_i;
        return sexprTail_type;
        break;
    case ELSE:
        sexprTail_type = sexprTail_i;
        return sexprTail_type;
        break;
    default:
        fprintf(listing, "SYNERR: Expecting addop, relop, ; ,  end, do, then, ] , , ,
) , or else. Received %s\n", tok->tokenname);
        while (tok->type != ENDFILE && tok->type != RELOP && tok->type != SEMICOLON &&
tok->type != END && tok->type != DO && tok->type != THEN && tok->type != CB &&
tok->type != COMMA && tok->type != CP && tok->type != ELSE)
        {
            tok = gettoken();
        }
        sexprTail_type = ERRSTAR;
        return sexprTail_type;
    }
    return ERR;
}
```

This specific function has an inherited attribute secprTail_i. In this specific function, if the token type is not addop we need to round the cape. This means we use the synthesized attribute sexprTail_type equal to the inherited variable and return this synthesized attribute.

This is the end of my explanation of some of the many important pieces of code in projects 3 and 4.

Discussions and Conclusions:

These were the most difficult projects that I have ever completed. At times, I thought that I would never be able to finish them, but Dr. Shenoi said it best: “the best way to eat an elephant is one bite at a time.” These projects were definitely a large elephant, but this process was incredibly rewarding. It was a bittersweet feeling whenever I finally finished the projects because I had a good time figuring out how to complete each of the projects and I put in an insane amount of time into them. All in all, I am excited that I was able to complete the pascal compiler, and have definitely learned more about programming in this class then all other CS courses combined.

Bibliography

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

Appendix 1

Listing file 1

```
1.   program test (input, output);
 2.    var a : integer;
 3.    var b : real;
 4.    var c : array [1..2] of integer;
 5.    function fun1(x:integer; y:real;
 6.                     z:array [1..2] of integer;
 7.                     q: real) : integer;
 8.      var d: integer;
 9.      begin
 10.        a := 2;
 11.        z[a] := 4;
 12.        c[2] := 3;
 13.        fun1 := c[1]
 14.       end;
 15.     function fun2(x: integer; y: integer) : real;
 16.       var e: real;
 17.       function fun3(n: integer; z: real) : integer;
```

```
18.         var e: integer;
19.         begin
20.           a:= e;
21.           e:= c[e];
22.           fun3 := 3
23.         end;
24.       begin
25.         a:= fun1(x, e, c, b);
26.         x:= fun3(c[1], e);
27.         e := e + 4.44;
28.         a:= (a mod y) div x;
29.         while ((a >= 4) and ((b <= e)
30.                         or (not (a = c[a])))) do
31.           begin
32.             a:= c[a] + 1
33.           end;
34.         fun2 := 2.5
35.       end;
36.   begin
37.     b:= fun2(c[4], c[5]);
38.     b:= fun2(c[4],2);
39.     if (a < 2) then a:= 1 else a := a + 2;
40.     if (b > 4.2) then a := c[a]
41.   end.
```

Address File 1

```
a loc0
b loc4
c loc12
d loc0
e loc0
e loc0
```

Listing file 2

```
1.  program test (input, output);
2.   var a : integer;
3.   var b : real;
4.   var c : array [1..2] of integer;
5.   var b : real;
 Duplicate variable name in scope
6.   function fun1(x:integer; y:real;
7.                  z:array [1..2] of integer;
8.                  q: real) : integer;
9.     var d: integer;
10.      begin
```

```
11.        a := 2;
12.        z[a] := 4;
13.        c[2] := 3;
14.        fun1 := c[1]
15.       end;
16.     function fun2(x: integer; y: integer) : real;
17.       var e: real;
18.       var e: integer;
 Duplicate variable name in scope
19.       function fun2(n: integer; z: real) : integer;
 Duplicate variable/function name in scope
20.         var e: integer;
 Duplicate variable name in scope
21.         begin
22.           a:= e;
 SEMERR: Unable to assign mixed types to each other
23.           e:= c[e];
 SEMERR: array var does not exist
24.           fun3 := 3
 SEMERR: Variable not declared
25.         end;
 SEMERR: Unable to assign mixed types to each other
26.       begin
27.         a:= fun1(x, e, c, b);
 SEMERR: Variable not declared
SEMERR: Variable not declared
SEMERR: Parameters entered do not match the function parameters
28.         x:= fun3(c[1], e);
 SEMERR: Variable not declared
SEMERR: Variable not declared
SEMERR: Variable not declared
SEMERR: Parameters entered do not match the function parameters
29.         e := e + 4.44;
 SEMERR: Variable not declared
SEMERR: Variable not declared
SEMERR: Cannot complete addop func on different types
30.         a:= (a mod y) div x;
 SEMERR: Variable not declared
SEMERR: Cannot complete mathematical operations with different typesSEMERR: Variable not
declared
31.         while ((a >= 4) and ((b <= e)
 SEMERR: Variable not declared
SEMERR: Unable to compare different types
32.                         or (not (a = c[a])))) do
```

```
33.           begin
34.             a:= c[a] + 1
35.           end;
36.         fun2 := 2.5
37.       end;
 SEMERR: Unable to assign mixed types to each other
38.    begin
39.      b:= fun2(c[4], c[5]);
 SEMERR: Parameters entered do not match the function parameters
SEMERR: Unable to assign mixed types to each other
40.      b:= fun2(c[4],2);
 SEMERR: Parameters entered do not match the function parameters
SEMERR: Unable to assign mixed types to each other
41.      if (a < 2) then a:= 1 else a := a + 2;
42.      if (b > 4.2) then a := c[a]
43.    end.
```

Address File 2

```
a loc0
b loc4
c loc12
b loc20
d loc0
e loc0
e loc8
e loc0
```

Listing File 3

```
1.    program test (input, output);
 2.     var a : integer;
 3.     var b : real;
 4.     var c : array [1..2] of integer;
 5.     var d : real;
 6.     function fun1(x:integer; y:real;
 7.                    z:array [1..2] of integer;
 8.                    q: real) : integer;
 9.       var d: integer;
 10.        begin
 11.          a := 2;
 12.          z[a] := 4;
 13.          c[2] := 3;
 14.          fun1 := c[1]
 15.        end;
 16.      function fun2(x: integer; y: integer) : real;
 17.        var e: real;
 18.        var f: integer;
```

```
19.        function fun3(n: integer; z: real) : integer;
20.          var e: integer;
21.          begin
22.            a:= e;
23.            e:= c[e];
24.            fun3 := 3
25.          end;
26.        begin
27.          a:= fun1(x, e, c, b);
28.          x:= fun3(c[1], e);
29.          e := e + 4.44;
30.          a:= (a mod y) div x;
31.          while ((a >= 4) and ((b <= e)
32.                          or (not (a = c[a])))) do
33.            begin
34.              a:= c[a] + 1
35.            end;
36.          fun2 := 2.5
37.        end;
38.    begin
39.      b:= fun2(c[4], c[5]);
40.      b:= fun2(c[4],2);
41.      if (a < 2) then a:= 1 else a := a + 2;
42.      if (b > 4.2) then a := c[a]
43.    end.
```

Address File 3

```
a loc0
b loc4
c loc12
d loc20
d loc0
e loc0
f loc8
e loc0
```

Listing File 4

```
1.    program test (input, output);
2.     var a : integer;
3.     var b : real;
4.     var c : array [1..2] of integer;
5.     var d : real;
6.     function fun1(x:integer; y:real;
7.                   z:array [1..2] of integer;
8.                   q: real) : integer;
9.       var d: integer;
```

```
10.      begin
11.        a := 2;
12.        z[a] := 4;
13.        c[2] := 3;
14.        fun1 := c[1]
15.      end;
16.    function fun2(x: integer; y: integer) : real;
17.      var e: real;
18.      var f: integer;
19.      function fun3(n: integer; z: real) : integer;
20.        var e: integer;
21.        begin
22.          a:= e;
23.          e:= c[e];
24.          fun3 := 3
25.        end;
26.      begin
27.        a:= fun1(x, e, c, b);
28.        x:= fun3(c[1], e);
29.        e := e + 4.44;
30.        a:= (a mod y) div x;
31.        while ((a >= 4) and ((b <= e)
32.                        or (not (a = c[a])))) do
33.          begin
34.            a:= c[a] + 1
35.          end;
36.        fun2 := 2.5
37.      end;
38.  begin
39.    b:= fun2(c[4], c[5]);
40.    b:= fun2(c[4], c[5,5]);
 SYNERR: Expecting mulop, addop, relop, ; ,  end, do, then, ] , , , ) , or else. Received NUM
SYNERR: Expecting comma or ) . Received CB
SEMERR: Actual function parameters do not match formal parameters
41.    b:= fun2(c[4], c[5], 5);
 SEMERR: Actual function parameters do not match formal parameters
42.    b:= fun2(c[4], c);
 SEMERR: Actual function parameters do not match formal parameters
43.    b:= fun2(c[4]);
 SEMERR: Actual function parameters do not match formal parameters
44.    b:= fun2();
 SYNERR: Expecting num, (, not, id, +, or - . Received CP
45.    b:= fun2;
 SEMERR: Actual function parameters do not match formal parameters
```

```
46.      b:= fun3(c[4], c[5]);
 SEMERR: Variable not declared
SEMERR: Unable to assign mixed types to each other
47.      b:= fun4(c[4], c[5]);
 SEMERR: Variable not declared
SEMERR: Unable to assign mixed types to each other
48.      b:= fun5(c[4], 2);
 SEMERR: Variable not declared
SEMERR: Unable to assign mixed types to each other
49.      if(a < 2) then a:= 1 else a:= a+2;
50.      if(b > 4.2) then a:= c[a]
51.    end.
```

Address File 4

```
a loc0
b loc4
c loc12
d loc20
d loc0
e loc0
f loc8
e loc0
```

Listing File 5

```
1.    program test (input, output);
2.     var a : integer;
3.     var b : real;
4.     var c : array [1..2] of integer;
5.     var d : real;
6.     function fun1(x:integer; y:real;
7.                    z:array [1..2] of integer;
8.                    q: real) : integer;
9.       var d: integer;
10.       begin
11.         a := 2;
12.         z[a] := 4;
13.         c[2] := 3;
14.         fun1 := c[1]
15.        end;
16.      function fun2(x: integer; y: integer) : real;
17.        var e: real;
18.        var f: integer;
19.        function fun3(n: integer; z: real) : integer;
20.          var e: integer;
21.          begin
22.            a:= e;
```

```
23.          e:= c[e];
24.          fun3 := 3
25.        end;
26.      begin
27.        a:= fun1(x, e, c, b);
28.        x:= fun3(c[1], e);
29.        e := e + 4;
 SEMERR: Cannot complete addop func on different types
30.        a:= (a mod 4.4) div 4.4;
 SEMERR: Cannot complete mathematical operations with different types
31.        while ((a >= 4.4) and ((b <= e)
 SEMERR: Unable to compare different types
32.                    or (not (a = c[a])))) do
33.          begin
34.            a:= c + 1.00 + $
LEXERR: Leading 0 after decimal   1.00
LEXERR: Unknown symbol:     $
35.          end;
36.        fun2 := 2.5
37.      end;
38.   begin
39.     b:= fun2(c[4], c[5]);
40.     b:= fun2(c[4], 2);
41.     if(a < 2) then a:= 1 else a:= a+2;
42.     if(b > 4.2) then a:= c[a]
43.   end.
```

Listing File 6

```
1.   program test (input, output);
2.    var a : integer;
3.    var b : real;
4.    var c : array [1..2] of integer;
5.    var d : real;
6.    function fun1(x:integer; y:real;
7.                    z:array [1..2] of integer;
8.                    q: real) : integer;
9.      var d: integer;
10.       begin
11.         a := 2;
12.         z[a] := 4;
13.         c[2] := 3;
14.         fun1 := c[1]
15.       end;
16.     function fun2(x: integer; y: integer) : real;
17.       var e: real;
```

```
18.       var f: integer;
19.       function fun3(n: integer; z: real) : integer;
20.         var e: integer;
21.         begin
22.           a:= e;
23.           e:= c[e];
24.           fun3 := 3
25.         end;
26.       begin
27.         a:= fun1(x, e, c, b);
28.         x:= fun3(c[1], e);
29.         e := e + 4;
 SEMERR: Cannot complete addop func on different types
30.         a:= (a mod 4.4) div 4.4;
 SEMERR: Cannot complete mathematical operations with different types
31.         while ((a >= 4.4) and ((b <= e)
 SEMERR: Unable to compare different types
32.                           or (not (a = c[a])))) do
33.           begin
34.             a:= c + 0.0
 SEMERR: Array format incorrect. Need to follow the strucutre ArrayID[integer]
SEMERR: Cannot complete addop func on different types
35.           end;
36.         fun2 := 2.5
37.       end;
38.   begin
39.     b:= fun2(c[4], c[5]);
40.     b:= fun2(c[4], 2);
41.     if(a < 2) then a := 1 else a:= a+2;
42.     if(b > 4.2) then a:= c[a]
43.   end.
```

Listing File 7

```
1.   program test (input, output);
2.    var a : integer;
3.    var b : real;
4.    var c : array [1..2] of integer;
5.    var d : real;
6.    function fun1(x:integer; y:real;
7.                    z:array [1..2] of integer;
8.                    q: real) : integer;
9.      var d: integer;
10.       begin
11.         a := 2;
12.         z[a] := 4;
```

```
13.        c[2] := 3;
14.        fun1 := c[1]
15.       end;
16.     function fun2(x: integer; y: integer) : real;
17.       var e: real;
18.       var f: integer;
19.       function fun3(n: integer; z: real) : integer;
20.         var e: integer;
21.         begin
22.           a:= e;
23.           e:= c[e];
24.           fun3 := 3
25.         end;
26.       begin
27.         a:= fun1(x, e, c, b);
28.         x:= fun3(c[1], e);
29.         e := e + 4;
 SEMERR: Cannot complete addop func on different types
30.         a:= (a mod 4.4) div 4.4;
 SEMERR: Cannot complete mathematical operations with different types
31.         while ((a >= 4.4) and ((b <= e)
 SEMERR: Unable to compare different types
32.                         or (not (a = c[a])))) do
33.           begin
34.             a:= c + 0
 SEMERR: Array format incorrect. Need to follow the strucutre ArrayID[integer]
  SEMERR: Cannot complete addop func on different types
35.           end;
36.         fun2 := 2.5
37.       end;
38.    begin
39.      b:= fun2(c[4], c[5]);
40.      b:= fun2(c[4], 2);
41.      if(a < 2) then a := 1 else a:= a+2;
42.      if(b > 4.2) then a:= c[a]
43.    end.
```

Appendix 2: