David Montgomery

CS4013

Project 1

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 lexical analyzer that would break an input text file down into different elements, or lexemes. We used eight separate machines to separate random input text into each lexeme. Each of these lexemes were given a meaning and will be used in the next three projects to develop the compiler.

Methodology:

As stated above, I used eight separate machines to break down input text into lexemes. Each of these machines are used to return the lexeme token that is being entered into the machine. The machines are a whitespace machine, reserved word machine, id machine, catchall machine, longreal machine, real machine, integer machine, and a relop machine. This section will be my mindset and mapping of the machines before tackling the coding aspect. Before I began to code, I first drew out how the machines would perform. The main goal of the white space machine is to iterate through spaces and tabs. So, if the character that was being passed into the white space machine was not a space or a tab, it would push it to the next machine. The next machine that I drew on paper was the reserved word machine. This machine is supposed to take a character, check if the character is a letter, and then iterate through until a character is not a letter. The main goal of this machine is to find words that are considered reserved words. Those reserved words are: program, var, real, function, begin, end, if, then, else, while, do, not, array, of, and integer. After the reserved word machine iterates through each character until the character is not a letter, it then will check if that word is equal to that of a reserved word. If it is not equal to a reserved word, then it should call the next machine. However, if it is equal to a reserved word, it should return that reserved word. The next machine that the reserved word machine calls is id word machine. An ID has to start with a letter, but is able to have numbers after the first character as well. An ID is basically any word that begins with a letter, but is not one of the reserved words in the previous machine. Another requirement for the ID machine is that the ID itself is less than 10 characters in length. So, when I code the machine, it needs to include error handling where if the ID is larger than 10 characters an error is returned. Similar to the reserved words, I will have a list of previously used IDs that the input file has written to the lexical analyzer. So, if you have used an ID that has previously been used, it will return that ID. However, if the ID in the input file has not been used, it will add it to the ID file and return the ID. If the first letter is not a letter, the ID machine will call the catchall machine. The next few machines are used if the character that is being inputted does not begin with a letter. The catchall machine will be used to catch all of the non-relop special characters. These characters are: (,) , , ; , : , [,] , + , - , * , .. , . , and := . These are special characters that our lexical analyzer needs to store and understand. If the input char is not one of these characters, that means one of two things. It is either a number or a relop. We begin by calling longreal. A longreal is a number with a whole part, a mantissa, and an

exponent. After drawing this machine on paper, the most difficult part is understanding the errors that are produced from this machine. Up until this point, no other machine is able to handle multiple errors. The ID machine needs to be able to handle the ID being greater than 10 characters, but that is it. This machine is able to handle any combination of five errors. Those errors being: the whole part of the number has a leading 0, the whole part of the number is greater than 5 numbers, the mantissa is greater than 5 numbers, the exponent part is greater than 2 numbers, and the exponent part has a leading zero. Unlike the other errors in this project, if the imputed number has any combination of these errors, we need to be able to return any or all of them. I will accomplish this by using flags which will be a global int that will notify if any combination of these errors happen. If the number does not have an exponent in it, then it is either an integer or a real number. I will first check if it's a real number by calling the real machine if the number does not include an exponent. I will handle the same errors in the same way as I did in the long real machine. If the number does not have a mantis, it will call the integer machine. The integer machine has to handle an additional error. The integer is not able to be greater than 10 characters. So, I will include that as a flag. If the character that is being checked is not a number, and has not been returned by any other machine. It has to either be a relop or an unknown symbol. The drawing for the relop machine is similar to that of the catchall machine. This machine will check if the inputted symbol is equal to any of the relop symbols. If they are equal, I will return the relop symbol. However, if they are not equal, this means that none of the machines caught the inputted char and, therefore, this is an unknown symbol. The relop machine will catch and return all relops and unknown symbols.

Implementation:

After concluding the drawings for the machines and understanding them, I begin to code the machines. I began this project by creating a main function. The main part of this main function is iterating through the input file. I use a while loop and the fgets function to iterate through the entire input file. I set a ptr equal to the buffer. The ptr is just a char * and the buffer is a buffer[72]. This buffer is equal to the first character of the first line of the input file. The following code is:

```
while(fgets(buffer, 72, input))
```

The next part of the main function is to detect when we finish each line. To do this, we create a while loop that loops through the first line until the ptr is either pointing at ‘\n’ or ‘\0’. This looks like this:

```
while((*ptr != ‘\n’) && (*ptr!=’\0’))
```

To get the tokens we create a variable of type token. In our header.h file, we create a struct called token that has a lex, tokenname, type, attname, att, and address. This struct looks like this:

```
struct token {
        char lex[20]
        char tokenname[10]
        int type
        char attname[10]
        union tokenattr {
                int att
                void* address
        }types;
        token* next;
};
```

This token that we create in our main function, tokGlob, is set equal to whitespace(&ptr). The whitespace machine is the first machine that is called and will begin the chain of calling the other machines. This code looks like this:

```
tokGlob = whitespace(&ptr)
```

The whitespace machine takes in the ptr, creates a char *front, and sets the front equal to the ptr. I then have a while loop that checks if the front is equal to either a space or a tab. If it is equal to one of these things, it increments the front by one and sets ptr equal to the front. I do this, so that no whitespace enters into the other machines. Also, if the front is equal to the end of the line or end of the file NULL is returned. If this happens, it breaks out of the while loop in the main function and will store the next line in the buffer, if applicable. The code is:

```
char *front = *ptr;
while(*front == ‘ ‘ || *front == ‘\t’) {
        front++
        *ptr = front
}
while(*front == ‘\n’ || *front == ‘\0’) {
        endOfLine = 1;
        return NULL;
}
```

Before using the next machine, I needed to create a linked list of all of the reserved words in the reserved.txt file. I use the same fgets function as I used in the main function. However, this time, I use a C function called sscanf. Sscanf is used to read formatted text from an input file. It then stores it into a new token. To create the linked list, I used sscanf along with creating a token called next. This next token stores the next line from the input file. This looks like this in code:

```
while(fgets(buffer, 72, reserved)) {
        sscanf(buffer, "%s %s %d %s %d", res->lex, res->tokenname, &res->type, res->attname,
&res->types.att);
res->next = malloc(sizeof(token))
res = res->next;
```

The next machine that whitespace calls is the reserved word machine. I created another *front and set it equal to the ptr. I use the C function isalpha to iterate through each letter and form a word. I create a while loop that loops until the next token is empty. In this loop I use the C function strncmp. Strncmp compares the word that I have produced using the isalpha, and the words in the reserved word linked list. If these are equal, I have created a switch statement that will return the reserved word. The code is as follows:

```
while(isalpha(*front)) {
        front++
}
while(res->next != NULL) {
        if(strncmp(res->lex,*ptr, front-*ptr)==0 && front-*ptr == strlen(res->lex))
```

If the word is not a reserved word, but still begins with a letter, it is an ID. So, if the if statement above is never true for any reserved word in the reserved word file, we call the id machine. Similar to the reserved word machine, the ID machine creates a linked list of words that have previously been used as IDs. This is done in nearly the same way as the reserved word linked list that I gave code for above. The only difference is it needs to be printed, so I have a while loop that loops until the linked list is empty and each id in the linked list is printed to an id list. The code is listed below:

```
while(id->next != NULL) {
        fprintf(reserved, "%s %s %d %s %d \n", id->lex, id-> tokenname, id->type, id->attname,
(int)id->types.address
        id = id->next
}
```

The main part of the actual ID machine begins with the same c function, isalpha, as the reserved word machine. This time, however, I check if this word that is produced is less than 10 characters. If it is greater than 10 characters, I return a LEXERR. If it is less than 10 characters, I check if it is already in the id word linked list. The id is returned if it is already in the list. If it is not in the list, we add it to the list and return the value. The code to check if the word is in the file can be seen below:

```
while(id->next != NULL) {
        if(strncmp(res->lex,*ptr, front-*ptr)==0 && front-*ptr == strlen(res->lex))
                {
                        *ptr = front;
                        return id
                }
        id = id->next
}
```

If the char that is being inputted is not a letter, the id machine will call the catchall machine. This machine is very simple, we just use a switch statement that will check if the ptr is equal to any of the catchall machine symbols. The only part that was not straightforward was the implementation of the dotdot symbol. Dot and dotdot are two separate symbols, so I created the case for dot and increment the ptr by one to see what the next symbol is. If the next symbol is another dot, we return dotdot. If it is not, we return the default case which would be dot. This can be seen below:

```
case '.':
        front++
        switch(*front)
        {
        case '.':
                strcpy(catchall->lex, "..")
                strcpy(catchall->tokenname, "DOTDOT")
                catchall->types = DOTDOT
                strcpy(catchall->attname, "NIL")
                catchall->types.att = NIL
                front++
                *ptr = front
                return catchall
        default:
                strcpy(catchall->lex, ".")
                strcpy(catchall->tokenname, "DOT")
                catchall->types = DOT
```

```
strcpy(catchall->attname, “NIL”)
catchall->types.att = NIL
*ptr = front
return catchall
```

The catchall machine will return the longreal machine if the ptr is a number. If it is a number, there are three different possibilities for the number: longreal, real, or integer. We test for the longreal first. If the ptr is a zero and has numbers after it, it is considered a leading zero. If this is the case, we iterate through the ptr until we no longer have a number, or we have reached five numbers before the decimal. If the whole part is greater than five numbers, I have a flag that is set equal to 1, so that we can check for other errors along with this one. This code can be seen here:

```
while(isdigit(*front) && (front-*ptr) < 5)
{
        front++
}
if(isdigit(*front))
{
        err3 = 1

        code continues...
}
```

The longreal machine consists of doing this multiple times: once for the whole part of the number, once for the mantissa, and once for the exponent. Each error is noted with a flag, and returned at the end of the function. If no errors occur, this means that we have 5 or less numbers in the whole part, 5 or less numbers in the mantissa, 2 or less numbers in the exponent, no leading zero in the whole part, and no leading zero in the exponent. If this is to happen, we return the number. This can be seen below:

```
 strncpy(realtoken->lex, *ptr, front-*ptr)
strcpy(realtoken->tokenname, “NUM”)
realtoken>types = NUM
strcpy(realtoken->attname, “LONGREAL”)
realtoken->types.att = LONGREAL
*ptr = front
return realtoken
```

The real machine is the same process as the longreal machine, except it does not include the exponent part of the number. The process of producing the real number is the same as what I have for the long real. If the number has no mantissa, the real machine will send it to the int machine. This can be seen below:

return intmachine(ptr)

In the intmachine, the process is similar to that of the longreal and real machine, however, there is a new error for the int being greater than 10 characters. This is handled as shown below:

```
while(isdigit(*front))
{
        front++
}
if((front-*ptr) > 10) {

        code continues…

}
```

If the ptr has not been returned by any of these machines, this means it is either a relop or an unknown symbol. The relop machine works in the same way as the catchall machine, the only difference is that it needs to handle unknown symbols. So, for this, we have a default case that can be seen below:

```
 strncpy(relop>lex, back , 1)
relop->lex[1] = '\0'
strcpy(relop>tokenname, "LEXERR")
relop->type = LEXERR
strcpy(relop>attname, "ERR1")
relop->types.att = ERR1
front++
*ptr = front
return relop
```

That is the last of the machines, so once something is returned to the token in the main function, we need to print this token and the errors, if applicable. Printing to the token file looks something like this:

fprintf(tokenoutput, “%d. %s %s %d %s %d \n”, i2, tokGlob->lex, tokGlob->tokenname, tokGlob->type, tokGlob->attname, tokGlob->types.att

We then check if we have any errors, and print the errors if we have any. This can be done like so:

```
if(tokGlob->type == LEXERR)
{

    errors are printed here…

}
```

The last step is to print the eof token. This is a token that is printed when we reach the end of the file. Once we break out of the initial while loop, we create a token and give it the eof attributes. This can be seen below:

```
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
```

This completes project 1.

Discussion and Conclusions

Overall, this project was incredibly challenging, but it was also very rewarding. The most difficult machine for myself was the id machine and the longreal machine. However, once these machines were completed the rest of the project, all though still lengthy and difficult, were easier to code. Now that I have completed the lexical analyzer, I have a better understanding of C programming, compilers, and computer science ideology.

References

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

Appendix 1

Input file 1

```
3.4E+1.
3E45.E+
3.E.+34
34.E25
00.05
&@
```

Token file 1

```
1.    3.4E+1 NUM 55 LONGREAL 47
1.    . DOT 20 NIL 50
2.    3 NUM 55 INT 23
2.    E45 ID 12 ADDRESS 759170704
2.    . DOT 20 NIL 50
2.    E ID 12 ADDRESS -2143287664
2.    + PLUS 48 NIL 50
3.    3 NUM 55 INT 23
3.    . DOT 20 NIL 50
3.    E ID 12 ADDRESS -2143287664
3.    . DOT 20 NIL 50
3.    + PLUS 48 NIL 50
3.    34 NUM 55 INT 23
4.    34 NUM 55 INT 23
4.    . DOT 20 NIL 50
4.    E25 ID 12 ADDRESS 340795712
5.    00.05 LEXERR 999 ERR2 902
6.    & LEXERR 999 ERR1 901
6.    @ LEXERR 999 ERR1 901
6.   EOF ENDFILE 49 NIL 50
```

Listing File 1

1. 3.4E+1.
2. 3E45.E+
3. 3.E.+34
4. 34.E25
5. 00.05

LEXERR: Leading Zero before decimal 00.05

6. &@

LEXERR: Unknown symbol: &

LEXERR: Unknown symbol: @

Input file 2

```
program test (input, output);
   this is a test
   74727
   7.3
```

Token file 2

1. program PROG 11 NIL 50
1. test ID 12 ADDRESS 1069549216
1. (OP 13 NIL 50
1. input ID 12 ADDRESS -490731888
1. , COMMA 14 NIL 50
1. output ID 12 ADDRESS -490731872
1.) CP 15 NIL 50
1. ; SEMICOLON 16 NIL 50
2. this ID 12 ADDRESS -1210052352
2. is ID 12 ADDRESS 1304441648
2. a ID 12 ADDRESS -490714288
2. test ID 12 ADDRESS 1069549216
3. 74727 NUM 55 INT 23
4. 7.3 NUM 55 REAL 24
4. EOF ENDFILE 49 NIL 50

Listing file 2

```
1.  program test (input, output);
2.      this is a test
3.      74727
4.      7.3
```

Appendix 2:

```c
#include "header.h"

void reservedList()
{

    char buffer2[72];
    FILE *reserved = fopen("reserved.txt", "r");

    token *res = malloc(sizeof(token));
    tokenList = res;

    while (fgets(buffer2, 72, reserved))
    {
        sscanf(buffer2, "%s %s %d %s %d", res->lex, res->tokenname, &res->type,
res->attname, &res->types.att);
        res->next = malloc(sizeof(token));
        res = res->next;
    }

    res = tokenList;
}

void linkedId()
{

    char buffer2[72];
    FILE *reserved = fopen("idwords.txt", "r");
```

```
    token *id = malloc(sizeof(token));
    idList = id;

    while (fgets(buffer2, 72, reserved))
    {
        sscanf(buffer2, "%s %s %d %s %d /n", id->lex, id->tokenname, &id->type,
id->attname, (int *)&id->types.address);
        id->next = malloc(sizeof(token));
        id = id->next;
    }

    id = idList;
    while (id->next != NULL)
    {
        fprintf(reserved, "%s %s %d %s %d \n", id->lex, id->tokenname, id->type,
id->attname, (int)id->types.address);
        id = id->next;
    }
}

token *relopmachine(char **ptr)
{
    char *front = *ptr;
    char *back = *ptr;
    token *relop = malloc(sizeof(token));

    switch (*front)
    {
    case '<':
        front++;
        switch (*front)
        {
        case '=':
            strcpy(relop->lex, "<=");
            strcpy(relop->tokenname, "LE");
            relop->type = LE;
            strcpy(relop->attname, "RELOP");
            relop->types.att = RELOP;
            front++;
            *ptr = front;
            return relop;
```

```
        case '>':
            strcpy(relop->lex, "<>");
            strcpy(relop->tokenname, "NEQ");
            relop->type = NEQ;
            strcpy(relop->attname, "RELOP");
            relop->types.att = RELOP;
            front++;
            *ptr = front;
            return relop;

        default:
            strcpy(relop->lex, "<");
            strcpy(relop->tokenname, "LT");
            relop->type = LT;
            strcpy(relop->attname, "RELOP");
            relop->types.att = RELOP;
            //front--;
            *ptr = front;
            return relop;
        }
    case '=':
        strcpy(relop->lex, "=");
        strcpy(relop->tokenname, "EQ");
        relop->type = EQ;
        strcpy(relop->attname, "RELOP");
        relop->types.att = RELOP;
        front++;
        *ptr = front;
        return relop;
    case '>':
        front++;
        switch (*front)
        {
        case '=':
            strcpy(relop->lex, ">=");
            strcpy(relop->tokenname, "GE");
            relop->type = GE;
            strcpy(relop->attname, "RELOP");
            relop->types.att = RELOP;
            front++;
```

```
            *ptr = front;
            return relop;

        default:
            strcpy(relop->lex, ">");
            strcpy(relop->tokenname, "GT");
            relop->type = GT;
            strcpy(relop->attname, "RELOP");
            relop->types.att = RELOP;
            *ptr = front;
            return relop;
        }

    default:

        strncpy(relop->lex, back, 1);
        relop->lex[1] = '\0';
        strcpy(relop->tokenname, "LEXERR");
        relop->type = LEXERR;
        strcpy(relop->attname, "ERR1");
        relop->types.att = ERR1;
        front++;
        *ptr = front;
        return relop;
    }
}

token *intmachine(char **ptr)
{
    err2 = 0;
    err3 = 0;
    err4 = 0;
    err5 = 0;
    err6 = 0;
    char *front = *ptr;
    char *back = *ptr;
    token *intToken = malloc(sizeof(token));

    front++;
    char leadingZero = *front;
    front--;
```

```
if (*front == '0' && isdigit(leadingZero))
{
    err2 = 1;
    while (isdigit(*front))
    {
        front++;
    }
    if ((front - *ptr) > 10)
    {
        err7 = 1;

        strncpy(intToken->lex, *ptr, front - *ptr);
        strcpy(intToken->tokenname, "LEXERR");
        intToken->type = LEXERR;
        strcpy(intToken->attname, "ERR7");
        intToken->types.att = ERR7;
        *ptr = front;

        return intToken;
    }

    strncpy(intToken->lex, *ptr, front - *ptr);
    strcpy(intToken->tokenname, "LEXERR");
    intToken->type = LEXERR;
    strcpy(intToken->attname, "ERR2");
    intToken->types.att = ERR2;
    *ptr = front;

    return intToken;
}
else if (isdigit(*front))
{
    while (isdigit(*front))
    {
        //printf("This is a number");
        front++;
    }
    if ((front - *ptr) > 10)
    {
        err7 = 1;
```

```
            strncpy(intToken->lex, *ptr, front - *ptr);
            strcpy(intToken->tokenname, "LEXERR");
            intToken->type = LEXERR;
            strcpy(intToken->attname, "ERR7");
            intToken->types.att = ERR7;
            *ptr = front;
            return intToken;
        }
        else
        {
            strncpy(intToken->lex, *ptr, front - *ptr);
            strcpy(intToken->tokenname, "NUM");
            intToken->type = NUM;
            strcpy(intToken->attname, "INT");
            intToken->types.att = INT;
            *ptr = front;
            return intToken;
        }
    }
    return relopmachine(ptr);
}

token *realmachine(char **ptr)
{
    err2 = 0;
    err3 = 0;
    err4 = 0;
    err5 = 0;
    err6 = 0;

    char *front = *ptr;
    //char *back = *ptr;
    token *realToken = malloc(sizeof(token));
    front++;
    char leadingZero = *front;
    front--;

    if (*front == '0' && isdigit(leadingZero))
    {
        err2 = 1;
        while (isdigit(*front) && (front - *ptr) < 5)
```

```
        {
            front++;
        }
        if (isdigit(*front))
        {
            err3 = 1;

            while (isdigit(*front))
            {
                front++;
            }
        }
        if (*front == '.')
        {
            front++;

            char *EDback2 = front;
            char *EDtemp2 = front;

            while (isdigit(*EDback2) && (EDback2 - EDtemp2) < 5)
            {
                EDback2++;
                front++;
            }

            if (isdigit(*front))
            {
                err4 = 1;

                while (isdigit(*front))
                {
                    front++;
                }

                strncpy(realToken->lex, *ptr, front - *ptr);
                strcpy(realToken->tokenname, "LEXERR");
                realToken->type = LEXERR;
                strcpy(realToken->attname, "ERR8");
                realToken->types.att = ERR8;
                *ptr = front;
                return realToken;
```

```
            }
            else
            {
                strncpy(realToken->lex, *ptr, front - *ptr);
                strcpy(realToken->tokenname, "LEXERR");
                realToken->type = LEXERR;
                strcpy(realToken->attname, "ERR2");
                realToken->types.att = ERR2;
                *ptr = front;
                return realToken;
            }
        }
        else
        {
            return intmachine(ptr);
        }
    }
    else if (*front == '0')
    {
        return intmachine(ptr);
    }
    else if (isdigit(*front))
    {
        while (isdigit(*front) && (front - *ptr) < 5)
        {
            front++;
        }
        if (isdigit(*front))
        {
            err3 = 1;

            while (isdigit(*front))
            {
                front++;
            }

            if (*front == '.')
            {
                front++;

                char *EDback2 = front;
```

```
            char *EDtemp2 = front;

            while (isdigit(*EDback2) && (EDback2 - EDtemp2) < 5)
            {
                EDback2++;
                front++;
            }

            if (isdigit(*front))
            {
                err4 = 1;

                while (isdigit(*front))
                {
                    front++;
                }
            }
            else
            {
                strncpy(realToken->lex, *ptr, front - *ptr);
                strcpy(realToken->tokenname, "LEXERR");
                realToken->type = LEXERR;
                strcpy(realToken->attname, "ERR2");
                realToken->types.att = ERR2;
                *ptr = front;
                return realToken;
            }
        }
        else
        {
            return intmachine(ptr);
        }
    }

    else if (*front == '.')
    {
        front++;
        if (isdigit(*front))
        {
            char *EDback4 = front;
            char *EDtemp4 = front;
```

```
            while (isdigit(*EDback4) && (EDback4 - EDtemp4) < 5)
            {
                EDback4++;
                front++;
            }
        }
        else
        {
            return intmachine(ptr);
        }
        if (isdigit(*front))
        {
            err4 = 1;

            while (isdigit(*front))
            {
                front++;
            }
            strncpy(realToken->lex, *ptr, front - *ptr);
            strcpy(realToken->tokenname, "LEXERR");
            realToken->type = LEXERR;
            strcpy(realToken->attname, "ERR8");
            realToken->types.att = ERR8;
            *ptr = front;
            return realToken;
        }

        else
        {
            strncpy(realToken->lex, *ptr, front - *ptr);
            strcpy(realToken->tokenname, "NUM");
            realToken->type = NUM;
            strcpy(realToken->attname, "REAL");
            realToken->types.att = REAL;
            *ptr = front;
            return realToken;
        }
    }
    else if (*front == '\0' || isblank(*front))
    {
        return intmachine(ptr);
```

```
        }
    }

    return intmachine(ptr);
}

token *longrealmachine(char **ptr)
{
    char *front = *ptr;
    //char *back = *ptr;
    token *realToken = malloc(sizeof(token));
    front++;
    char leadingZero = *front;
    front--;

    if (*front == '0' && isdigit(leadingZero))
    {
        err2 = 1;
        while (isdigit(*front) && (front - *ptr) < 5)
        {
            front++;
        }
        if (isdigit(*front))
        {
            err3 = 1;

            while (isdigit(*front))
            {
                front++;
            }
        }
        if (*front == '.')
        {
            front++;

            char *EDback2 = front;
            char *EDtemp2 = front;

            while (isdigit(*EDback2) && (EDback2 - EDtemp2) < 5)
            {
                EDback2++;
```

```
            front++;
        }

        if (isdigit(*front))
        {
            err4 = 1;

            while (isdigit(*front))
            {
                front++;
            }
            if (isdigit(*front))
            {
                return realmachine(ptr);
            }
        }
        if (*front == 'e' || *front == 'E')
        {
            front++;

            if (*front == '-' || *front == '+')
            {
                front++;
            }

            if (*front == '0')
            {
                err5 = 1;
            }

            char *EDback3 = front;
            char *EDtemp3 = front;
            while (isdigit(*EDback3) && (EDback3 - EDtemp3) < 2)
            {
                EDback3++;
                front++;
            }
            if (isdigit(*front))
            {
                err6 = 1;
            }
```

```
            while (isdigit(*front))
            {
                front++;
            }

            strncpy(realToken->lex, *ptr, front - *ptr);
            strcpy(realToken->tokenname, "LEXERR");
            realToken->type = LEXERR;
            strcpy(realToken->attname, "ERR2");
            realToken->types.att = ERR2;
            *ptr = front;
            return realToken;
        }
    }
    else
    {
        return realmachine(ptr);
    }
}
else if (*front == '0')
{
    return intmachine(ptr);
}
else if (isdigit(*front))
{
    while (isdigit(*front) && (front - *ptr) < 5)
    {
        front++;
    }
    if (isdigit(*front))
    {
        err3 = 1;

        while (isdigit(*front))
        {
            front++;
        }

        if (*front == '.')
        {
```

```
                front++;

                char *EDback2 = front;
                char *EDtemp2 = front;

                while (isdigit(*EDback2) && (EDback2 - EDtemp2) < 5)
                {
                    EDback2++;
                    front++;
                }

                if (isdigit(*front))
                {
                    err4 = 1;

                    while (isdigit(*front))
                    {
                        front++;
                    }
                    if (isdigit(*front))
                    {
                        return realmachine(ptr);
                    }
                }
                if (*front == 'e' || *front == 'E')
                {
                    front++;

                    if (*front == '-' || *front == '+')
                    {
                        front++;
                    }

                    if (*front == '0')
                    {
                        err5 = 1;
                    }

                    char *EDback3 = front;
                    char *EDtemp3 = front;
                    while (isdigit(*EDback3) && (EDback3 - EDtemp3) < 2)
```

```
                {
                    EDback3++;
                    front++;
                }
                if (isdigit(*front))
                {
                    err6 = 1;
                }

                while (isdigit(*front))
                {
                    front++;
                }

                strncpy(realToken->lex, *ptr, front - *ptr);
                strcpy(realToken->tokenname, "LEXERR");
                realToken->type = LEXERR;
                strcpy(realToken->attname, "ERR8");
                realToken->types.att = ERR8;
                *ptr = front;
                return realToken;
            }
        }
        else
        {
            return realmachine(ptr);
        }
    }

    else if (*front == '.')
    {
        front++;
        if (isdigit(*front))
        {
            char *EDback4 = front;
            char *EDtemp4 = front;
            while (isdigit(*EDback4) && (EDback4 - EDtemp4) < 5)
            {
                EDback4++;
                front++;
            }
```

```
            }
            else
            {
                return realmachine(ptr);
            }
            if (isdigit(*front))
            {
                err4 = 1;

                while (isdigit(*front))
                {
                    front++;
                }

                if (*front == 'e' || *front == 'E')
                {
                    front++;

                    if (*front == '+' || *front == '-')
                    {
                        front++;
                    }
                    if (*front == '0')
                    {
                        err5 = 1;
                    }
                    char *EDback5 = front;
                    char *EDtemp5 = front;
                    while (isdigit(*EDback5) && (EDback5 - EDtemp5) < 2)
                    {
                        EDback5++;
                        front++;
                    }
                    if (isdigit(*front))
                    {
                        err6 = 1;

                        while (isdigit(*front))
                        {
                            front++;
                        }
```

```
                    strncpy(realToken->lex, *ptr, front - *ptr);
                    strcpy(realToken->tokenname, "LEXERR");
                    realToken->type = LEXERR;
                    strcpy(realToken->attname, "ERR8");
                    realToken->types.att = ERR8;
                    *ptr = front;
                    return realToken;
                }
            }
            else
            {
                return realmachine(ptr);
            }
        }
        else if (*front == '\0' || isblank(*front))
        {
            return realmachine(ptr);
        }
        else if (*front == 'e' || *front == 'E')
        {
            front++;
            if (*front == '+' || *front == '-')
            {
                front++;
            }
            if (*front == '0')
            {
                err5 = 1;

                char *EDback6 = front;
                char *EDtemp6 = front;
                while (isdigit(*EDback6) && (EDback6 - EDtemp6) < 2)
                {
                    EDback6++;
                    front++;
                }
                if (isdigit(*front))
                {
                    err6 = 1;
                }
                while (isdigit(*front))
```

```
                {
                    front++;
                }
                strncpy(realToken->lex, *ptr, front - *ptr);
                strcpy(realToken->tokenname, "LEXERR");
                realToken->type = LEXERR;
                strcpy(realToken->attname, "ERR8");
                realToken->types.att = ERR8;
                *ptr = front;
                return realToken;
            }
            else
            {
                char *EDback7 = front;
                char *EDtemp7 = front;
                while (isdigit(*EDback7) && (EDback7 - EDtemp7) < 2)
                {
                    EDback7++;
                    front++;
                }
                if (isdigit(*front))
                {
                    while (isdigit(*front))
                    {
                        front++;
                    }
                    err6 = 1;
                    strncpy(realToken->lex, *ptr, front - *ptr);
                    strcpy(realToken->tokenname, "LEXERR");
                    realToken->type = LEXERR;
                    strcpy(realToken->attname, "ERR2");
                    realToken->types.att = ERR2;
                    *ptr = front;
                    return realToken;
                }
                else
                {
                    strncpy(realToken->lex, *ptr, front - *ptr);
                    strcpy(realToken->tokenname, "NUM");
                    realToken->type = NUM;
                    strcpy(realToken->attname, "LONGREAL");
```

```
                        realToken->types.att = LONGREAL;
                        *ptr = front;
                        return realToken;
                    }
                }
            }
        }
    }
    return realmachine(ptr);
}

token *catchallmachine(char **ptr)
{
    char *front = *ptr;
    char *back = *ptr;
    token *catchall = malloc(sizeof(token));

    switch (*front)
    {
    case '(':
        strcpy(catchall->lex, "(");
        strcpy(catchall->tokenname, "OP");
        catchall->type = OP;
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
    case ')':
        strcpy(catchall->lex, ")");
        strcpy(catchall->tokenname, "CP");
        catchall->type = CP;
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
    case ',':
        strcpy(catchall->lex, ",");
        strcpy(catchall->tokenname, "COMMA");
        catchall->type = COMMA;
```

```
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
    case ';':
        strcpy(catchall->lex, ";");
        strcpy(catchall->tokenname, "SEMICOLON");
        catchall->type = SEMICOLON;
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
    case ':':
        strcpy(catchall->lex, ":");
        strcpy(catchall->tokenname, "COLON");
        catchall->type = COLON;
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
    case '[':
        strcpy(catchall->lex, "[");
        strcpy(catchall->tokenname, "OB");
        catchall->type = OB;
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
    case ']':
        strcpy(catchall->lex, "]");
        strcpy(catchall->tokenname, "CB");
        catchall->type = CB;
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
```

```
    case '+':
        strcpy(catchall->lex, "+");
        strcpy(catchall->tokenname, "PLUS");
        catchall->type = PLUS;
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
    case '-':
        strcpy(catchall->lex, "-");
        strcpy(catchall->tokenname, "MINUS");
        catchall->type = MINUS;
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
    case '*':
        strcpy(catchall->lex, "*");
        strcpy(catchall->tokenname, "MULTIPLY");
        catchall->type = MULTIPLY;
        strcpy(catchall->attname, "NIL");
        catchall->types.att = NIL;
        front++;
        *ptr = front;
        return catchall;
    case '.':
        front++;
        switch (*front)
        {
        case '.':
            strcpy(catchall->lex, "..");
            strcpy(catchall->tokenname, "DOTDOT");
            catchall->type = DOTDOT;
            strcpy(catchall->attname, "NIL");
            catchall->types.att = NIL;
            front++;
            *ptr = front;
            return catchall;
        default:
```

```
            strcpy(catchall->lex, ".");
            strcpy(catchall->tokenname, "DOT");
            catchall->type = DOT;
            strcpy(catchall->attname, "NIL");
            catchall->types.att = NIL;
            *ptr = front;
            return catchall;
        }
    default:
        return longrealmachine(ptr);
    }
}

token *idmachine(char **ptr)
{

    char buffer2[72];
    FILE *idwords = fopen("idwords.txt", "a+");

    token *id = malloc(sizeof(token));
    //idList = id;

    while (fgets(buffer2, 72, idwords))
    {
        sscanf(buffer2, "%s %s %d %s %d", id->lex, id->tokenname, &id->type,
id->attname, (int *)&id->types.address);
        id->next = malloc(sizeof(token));
        id = id->next;
    }
    id = idList;
    //id->next = malloc(sizeof(token));
    //id->next = NULL;
    char *front = *ptr;
    char *idptr = *ptr;

    if (!isalpha(*front))
    {
        //printf("Not a letter \n");
        return catchallmachine(ptr);
    }
    if (isalnum(*front))
```

```
    {
        while (isalnum(*front))
        {
            front++;
        }
        if ((front - *ptr) > 10)
        {
            while (isalnum(*front))
            {
                front++;
            }
            strncpy(id->lex, *ptr, front - *ptr);
            strcpy(id->tokenname, "LEXERR");
            id->type = LEXERR;
            strcpy(id->attname, "ERR3");
            id->types.att = ERR3;
            *ptr = front;
            return id;
        }

        while (id->next != NULL)
        {
            if (strncmp(id->lex, *ptr, front - *ptr) == 0 && front - *ptr ==
strlen(id->lex))
            {
                *ptr = front;
                test = 1;
                return id;
            }
            id = id->next;
        }
        id = idList;

        //id->next = NULL;

        //id = id->next;
        //id = id->next;
        while (id->next != NULL)
        {
            id = id->next;
        }
```

```
        strncpy(id->lex, *ptr, front - *ptr);
        strcpy(id->tokenname, "ID");
        id->type = ID;
        strcpy(id->attname, "ADDRESS");
        id->types.address = malloc(sizeof(void));

        //printf("its a match \n");

        fprintf(idwords, "%s %s %d %s %d \n", id->lex, id->tokenname, id->type,
id->attname, (int)id->types.address);
        id->next = malloc(sizeof(token));
        *ptr = front;
        id = idList;
        return id;
    }

    else
    {
        return catchallmachine(ptr);
    }
}

token *resmachine(char **ptr)
{
    token *res = malloc(sizeof(token));

    res = tokenList;

    char *front = *ptr;
    char *word = *ptr;

    token *rettoken = malloc(sizeof(token));

    if (!isalpha(*front))
    {
        //printf("Not a letter \n");
        return idmachine(ptr);
    }

    while (isalpha(*front))
    {
```

```
        front++;
    }
    while (res->next != NULL)
    {
        if (strncmp(res->lex, *ptr, front - *ptr) == 0 && front - *ptr ==
strlen(res->lex))
        {
            strncpy(rettoken->lex, *ptr, front - *ptr);
            switch (*word)
            {
            case 'v':
                //strcpy(rettoken->lex,"var");
                strcpy(rettoken->tokenname, "VAR");
                rettoken->type = VAR;
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                word++;
                *ptr = front;
                return rettoken;
            case 'p':
                strcpy(rettoken->tokenname, "PROG");
                rettoken->type = PROG;
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                word++;
                *ptr = front;
                res = res->next;
                return rettoken;
            case 'r':
                strcpy(rettoken->tokenname, "REAL");
                rettoken->type = REAL;
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                *ptr = front;
                return rettoken;
            case 'f':
                strcpy(rettoken->tokenname, "FUNCTION");
                rettoken->type = FUNCTION;
```

```
            strcpy(rettoken->attname, "NIL");
            rettoken->types.att = NIL;
            //printf("its a match \n");
            *ptr = front;
            return rettoken;
        case 'b':
            strcpy(rettoken->tokenname, "BEGIN");
            rettoken->type = BEGIN;
            strcpy(rettoken->attname, "NIL");
            rettoken->types.att = NIL;
            //printf("its a match \n");
            *ptr = front;
            return rettoken;
        case 'e':
            word++;
            switch (*word)
            {
            case 'l':
                strcpy(rettoken->tokenname, "ELSE");
                rettoken->type = ELSE;
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                *ptr = front;
                return rettoken;
            default:
                strcpy(rettoken->tokenname, "END");
                rettoken->type = END;
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                *ptr = front;
                return rettoken;
            }
        case 'i':
            word++;
            switch (*word)
            {
            case 'f':
                strcpy(rettoken->tokenname, "IF");
                rettoken->type = IF;
```

```
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                *ptr = front;
                return rettoken;
            default:
                strcpy(rettoken->tokenname, "INTEGER");
                rettoken->type = INTEGER;
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                *ptr = front;
                return rettoken;
            }
        case 't':
            strcpy(rettoken->tokenname, "THEN");
            rettoken->type = THEN;
            strcpy(rettoken->attname, "NIL");
            rettoken->types.att = NIL;
            //printf("its a match \n");
            *ptr = front;
            return rettoken;
        case 'w':
            strcpy(rettoken->tokenname, "WHILE");
            rettoken->type = WHILE;
            strcpy(rettoken->attname, "NIL");
            rettoken->types.att = NIL;
            //printf("its a match \n");
            *ptr = front;
            return rettoken;
        case 'd':
            strcpy(rettoken->tokenname, "DO");
            rettoken->type = DO;
            strcpy(rettoken->attname, "NIL");
            rettoken->types.att = NIL;
            //printf("its a match \n");
            *ptr = front;
            return rettoken;
        case 'n':
            strcpy(rettoken->tokenname, "NOT");
            rettoken->type = NOT;
```

```
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                *ptr = front;
                return rettoken;
            case 'a':
                strcpy(rettoken->tokenname, "ARRAY");
                rettoken->type = ARRAY;
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                *ptr = front;
                return rettoken;
            case 'o':
                strcpy(rettoken->tokenname, "OF");
                rettoken->type = OF;
                strcpy(rettoken->attname, "NIL");
                rettoken->types.att = NIL;
                //printf("its a match \n");
                *ptr = front;
                return rettoken;

            default:
                return catchallmachine(ptr);
            }
        }

        res = res->next;
    }
    return idmachine(ptr);
}

token *whitemachine(char **ptr)
{
    endOfLine = 0;
    char *front = *ptr;

    while (*front == ' ' || *front == '\t')
    {
        front++;
        *ptr = front;
```

```
    }

    while (*front == '\n' || *front == '\0')
    {
        endOfLine = 1;
        return NULL;
    }
    return resmachine(ptr);
}

int main()
{
    reservedList();
    linkedId();
    int count = 0;
    int after = 0;
    char *front;
    char *ptr;
    int i = 1;
    int i2 = 1;
    char buffer[72];
    token *tokGlob;
    FILE *input = fopen("input.txt", "r");
    FILE *tokenOutput = fopen("token.txt", "w");
    int firsterror = 0;

    FILE *listing = fopen("listing.txt", "w");

    while (fgets(buffer, 72, input))
    {
       // fprintf(listing, "\n");
        //after = 0;
        if (count > 0 )
        {
            fprintf(listing, "\n %d.    %s", i, buffer);
            //after = 1;
            i2++;
        }
        else
        {
            fprintf(listing, "%d.    %s", i, buffer);
```

```
        }
        ptr = buffer;

        i++;
        count++;

        while ((*ptr != '\n') && (*ptr != '\0'))
        {

            tokGlob = whitemachine(&ptr);
            if (endOfLine == 1)
            {
                break;
            }

            fprintf(tokenOutput, "%d.     %s %s %d %s %d \n", i2, tokGlob->lex,
tokGlob->tokenname, tokGlob->type, tokGlob->attname, tokGlob->types.att);

            if (tokGlob->type == LEXERR)
            {
                if (tokGlob->types.att == ERR1)
                {

                    if (firsterror == 0)
                    {
                        fprintf(listing, "LEXERR: Unknown symbol:     %s",
tokGlob->lex);
                    }
                    else
                    {
                        fprintf(listing, "\nLEXERR: Unknown symbol:     %s",
tokGlob->lex);
                    }
                    firsterror = 1;

                    //after = 0;
                    //count++;
                }
                if (tokGlob->types.att == ERR3)
                {
```

```
                    if (firsterror == 0)
                    {
                        fprintf(listing, "LEXERR: ID to large     %s",
tokGlob->lex);
                    }
                    else
                    {
                        fprintf(listing, "\nLEXERR: ID to large     %s",
tokGlob->lex);
                    }
                    firsterror = 1;

                    //after = 0;
                    //count++;
                }

                if (err2 == 1)
                {
                    if (firsterror == 0)
                    {
                        fprintf(listing, "LEXERR: Leading Zero before decimal
%s", tokGlob->lex);
                    }
                    else
                    {
                        fprintf(listing, "\nLEXERR: Leading Zero before decimal
%s", tokGlob->lex);
                    }
                    firsterror = 1;
                    //after = 0;
                    err2 = 0;
                    //count++;
                }
                if (err3 == 1)
                {
                    if (firsterror == 0)
                    {
                        fprintf(listing, "LEXERR: Number before decimal point is
greater than 5 characters    %s", tokGlob->lex);
                    }
                    else
```

```
                    {
                        fprintf(listing, "\nLEXERR: Number before decimal point is
greater than 5 characters    %s", tokGlob->lex);
                    }
                    firsterror = 1;

                    //after = 0;
                    err3 = 0;
                    //count++;
                }
                if (err4 == 1)
                {
                    if (firsterror == 0)
                    {
                        fprintf(listing, "LEXERR: Number after decimal point is
greater than 5 characters     %s", tokGlob->lex);
                    }
                    else
                    {
                        fprintf(listing, "\nLEXERR: Number after decimal point is
greater than 5 characters     %s", tokGlob->lex);
                    }
                    firsterror = 1;
                    err4 = 0;
                    //after = 0;
                    //count++;
                }
                if (err5 == 1)
                {
                    if (firsterror == 0)
                    {
                        fprintf(listing, "LEXERR: Leading zero in the exponenet
%s", tokGlob->lex);
                    }
                    else
                    {
                        fprintf(listing, "\nLEXERR: Leading zero in the exponenet
%s", tokGlob->lex);
                    }
                    firsterror = 1;
```

```
                    //after = 0;
                    err5 = 0;
                    //count++;
                }
                if (err6 == 1)
                {
                    if (firsterror == 0)
                    {
                        fprintf(listing, "LEXERR: Numbers after exponent exceed 2
characters  %s", tokGlob->lex);
                    }
                    else
                    {
                        fprintf(listing, "\nLEXERR: Numbers after exponent exceed 2
characters  %s", tokGlob->lex);
                    }
                    firsterror = 1;

                    //after = 0;
                    err6 = 0;
                    //count++;
                }
                if (err7 == 1)
                {
                    if (firsterror == 0)
                    {
                        fprintf(listing, "LEXERR: Int is greater than 10 digits
%s", tokGlob->lex);
                    }
                    else
                    {
                        fprintf(listing, "\nLEXERR: Int is greater than 10 digits
%s", tokGlob->lex);
                    }
                    firsterror = 1;

                    //after = 0;
                    err7 = 0;
                    //count++;
                }
                firsterror=0;
```

```
                after = 1;
            }
        }
    }

    token *eof = malloc(sizeof(struct token));
    strcpy(eof->lex, "EOF");
    strcpy(eof->tokenname, "ENDFILE");
    eof->type = ENDFILE;
    strcpy(eof->attname, "NIL");
    eof->types.att = NIL;
    fprintf(tokenOutput, "%d.    %s %s %d %s %d \n", i2, eof->lex, eof->tokenname,
eof->type, eof->attname, eof->types.att);
}
```