A typesetting system for mathematical expressions

(Final report for class "Principles and Techniques of Compiler Design")

Yichen Wang

May 31st, 2016

Abstract

In this project, I designed a typesetting system based on the provided instructions. This system can typeset basic math expressions. During the project, I managed to complete lexical analysis, syntax analysis and syntax-directed translation. The system is also able to discover errors at each step of analysis.

1 Introduction and project instruction

This typesetting system uses recursive descent method to conduct syntax analysis. Given the source text file, the system compiles, typesets and then generates an HTML file.

The instructions of this project are detailed in the last 3 pages, which describe the character set and definitions, language rules, basic grammar, input/output requirements and test samples.

2 The specific implementation of typesetting

2.1 Lexical analysis

According to the instructions, acceptable characters are:

Lowercases and uppercases; numbers; special characters: (,), \, {, }, _, $; separators: blanks and line feeds.

Identifier(id): begins with letters and consists of letters and numbers.

Number(num): unsigned integer.

Regular expressions are defined as follows:

$$letter \rightarrow a|b|...|z|A|B|...|Z$$
$$digit \rightarrow 0|1|...|9$$
$$other \rightarrow (|)|\{|\}|\ \hat{}\ |\_|\$$
$$N \rightarrow digit(digit)^*$$
$$i \rightarrow letter(letter|digit)^*$$
$$B \rightarrow \backslash blank$$
$$S \rightarrow \backslash sum$$
$$I \rightarrow \backslash int$$

where N is number(num), i is identifier(id), B is blank, S is sum, I is integral.

According to regular expressions, we can achieve the deterministic finite automation(DFA), which can identify number, identifier, fixed words($blank, sum, int$) and other legal characters. The DFA is shown in Fig.1, 2, 3, 4.

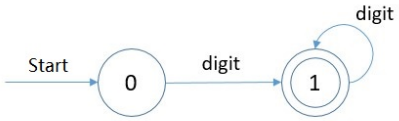


Fig. 1: DFA which identifies number

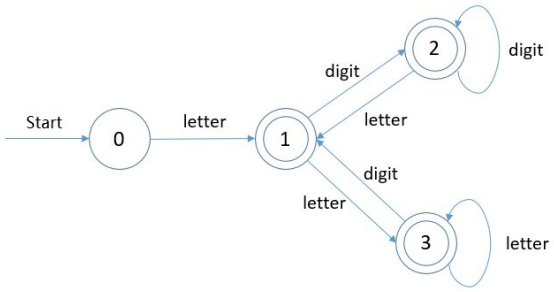


Fig. 2: DFA which identifies id

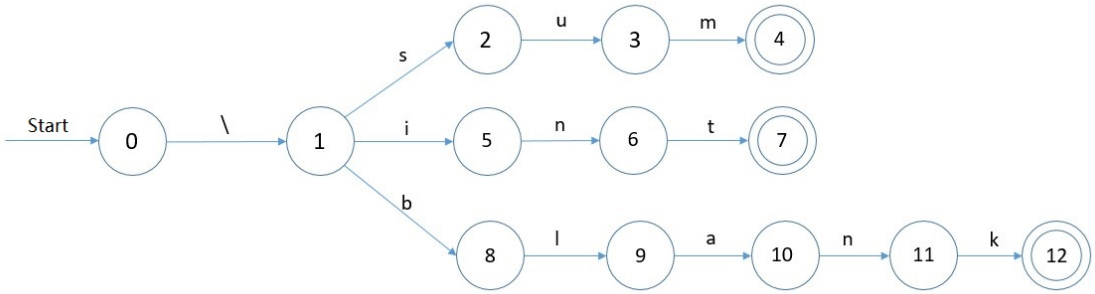


Fig. 3: DFA which identifies fixed words

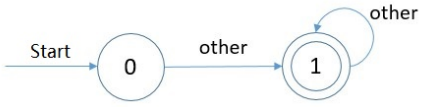


Fig. 4: DFA which identifies other legal characters

2.2 Syntax analysis

Recursive decent parsing in LL(1) analysis is applied in syntax analysis. The given grammar is not complex and with a small modification, there is no collision after eliminating left-recursion, extracting public left factors and disambiguation. Compared with LR analysis, this method is better because it requires less time and is less system resources consuming. Thus, I applied this method in my work.

2.2.1 Eliminating left-recursion

LL grammar shouldn't have left-recursion, but the following productions in the given grammar have left-recursion, which require modifying.

$$B \to BB$$
$$B \to B\_\ \hat{}\ BB$$
$$B \to B\ \hat{}\ B$$
$$B \to B\_B$$

Take a production $A \to A\alpha|\beta$ for example, which has left-recursion, the following 2 production can be used to eliminate left-recursion.

$$A \to \beta A'$$
$$A' \to \alpha|A'\epsilon$$

First, combine productions without left-recursion in the original grammar into one production, i.e. $B \to id|num|(B)|\backslash blank|\backslash int\{B\}\{B\}\{B\}|\backslash sum\{B\}\{B\}\{B\}$
and take it as β shown above. Then take B, $\hat{}\ \{B\}$, $\_\{B\}$, $\_\ \hat{}\ \{B\}$ which are in productions with left-recursion as α shown above and eliminate left-recursion, and we get the following productions:

$$S \to \$B\$$$
$$B \to TM$$
$$M \to UM|\epsilon$$
$$U \to\ \hat{}\ \{B\}|\_\ \hat{}\ \{B\}\{B\}|\_\{B\}|B$$
$$T \to id|num|(B)|\backslash blank|\backslash int\{B\}\{B\}\{B\}|\backslash sum\{B\}\{B\}\{B\}$$

2.2.2 Extracting public left factors

The above productions don't have left-recursion, but they have public left factor "_" and it needs to be extracted. Extract "_" from production $U \to\ \hat{}\ \{B\}|\_\ \hat{}\ \{B\}|\_\{B\}|B$ and change it to following:

$$U \to\ \hat{}\ \{B\}|\_V|B$$
$$V \to\ \hat{}\ \{B\}\{B\}|\{B\}$$

Then the grammar is adjusted to following:

$$S \to \$B\$$$
$$B \to TM$$
$$M \to UM|\epsilon$$
$$U \to\ \hat{}\ \{B\}|\_V|B$$
$$V \to\ \hat{}\ \{B\}\{B\}|\{B\}$$
$$T \to id|num|(B)|\backslash blank|\backslash int\{B\}\{B\}\{B\}|\backslash sum\{B\}\{B\}\{B\}$$

There is no left-recursion and public left factor in the grammar.

2.2.3 Disambiguation

Although left-recursion and public left factors are eliminated, this grammar is still not an LL(1) grammar because $FIRST(U) \cap FOLLOW(M) \neq \emptyset$. Thus there is ambiguation in this grammar. For example:

$$S \to \$B\$ \to TM \to TUM \to TBM \to TTMM \to$$
$$TTUMM \to TTBMM \to TTTMMM \to aaa$$
$$S \to \$B\$ \to TM \to TUM \to TU \to TB \to TTM \to$$
$$TTUM \to TTBM \to TTMMM \to aaa$$

This indicates that the grammar has ambiguity. Adjust the grammar and we get the final version, which is as follows:

$$S \to \$B\$$$
$$B \to TM$$
$$M \to UM|B|\epsilon$$
$$U \to \ \hat{}\ \{B\}|\_V$$
$$V \to \ \hat{}\ \{B\}\{B\}|\{B\}$$
$$T \to id|num|(B)|\backslash blank|\backslash int\{B\}\{B\}\{B\}|\backslash sum\{B\}\{B\}\{B\}$$

This grammar has the same implication as original one. Eliminating left-recursion and extracting public left factors doesn't change its implication. Disambiguation only moves B from U's production to M's.

2.2.4 Producing predictive parsing table

The process of producing predictive parsing table is as follows [1]:
(1)For each production $A \to \alpha$, run (2) and (3);
(2)For each terminal a in $FIRST(\alpha)$, add $A \to \alpha$ to M[A, a];
(3)If ϵ is in $FIRST(\alpha)$, for each terminal β(including \$) in $FOLLOW(A)$, add $A \to \alpha$ to M[A, β];
(4)Other items in M without definition are error entry.

For the following productions, the above algorithm produces the predictive parsing table, which is shown in Table.1.

$$1.\ S \to \$B\$$$
$$2.\ B \to TM$$
$$3.\ M \to UM\ \ 4.\ M \to B$$
$$5.\ M \to \epsilon$$
$$6.\ U \to \ \hat{}\ \{B\}$$
$$7.\ U \to \_V$$
$$8.\ V \to \ \hat{}\ \{B\}\{B\}$$
$$9.\ V \to \{B\}$$
$$10. T \to id$$
$$11. T \to num$$
$$12. T \to (B)$$

$$13.T \rightarrow \backslash blank$$
$$14.T \rightarrow \backslash int\{B\}\{B\}\{B\}$$
$$15.T \rightarrow \backslash sum\{B\}\{B\}\{B\}$$

Table 1: Predictive parsing table for LL analysis

Nonterminals	Input symbols											
	id	number	\blank	\sum	\int	(	)	{	}	_	^	$
S												1
B	2	2	2	2	2	2						
T	10	11	13	14	15	12						
M	4	4	4	4	4	4	5		5	3	3	5
U										7	6	
V								8			8	

2.3 syntax-directed translation(semantic analysis)

This is the last step of this project. Typesetting is done in this step and I need to figure out each character's exact location and size, which are represented by three parameters: horizontal and vertical coordinates and size. For HTML, the corresponding parameters are $left, top$ and $font-size$. The former two parameters is the horizontal and vertical distance to the upper left corner, and the last parameter is character's size. Use l, t, f to represent them and store the three variables in an array *location*. For superscript and subscript, like $\backslash int\{B\}\{B\}\{B\}$, the coordinates of the latter two Bs cannot be directly calculated by adding or substracting a number to the first B's coordinate. Specially, I use two functions *sup*() and *sub*() to calculate the superscript and subscript's coordinates. For typesetting, first set an initial value for the array *location*, which determines the expression's general location and size in the web page. Based on the initial value, I calculate the other parameters for each character.

It can be determined that l, t, f are inherited properties. In the following instructions I use *loca* to represent l, t, f and $mkleaf()$ is the function that calculates the termials' properties. From this we can get Table.2, which shows the specific syntax-directed translation.

2.4 Localization of errors

2.4.1 Lexical analysis

Illegal characters can be identified in lexical analysis. When DFA receives illegal characters, it prints them out with the their location and ends the analysis. The codes are shown in Fig.5, 6, in which "state" represents states after receiving different legal characters.

2.4.2 Syntax analysis

Recursive decent parsing is used in syntax analysis, so when there is an incorrect production, the program prints the incorrect production and returns -1, which represents an

Table 2: Specific syntax-directed translation

$S \rightarrow \$$ B $\$$	$B.loca = S.loca$
$B \rightarrow$ T M	$T.loca = B.loca$ $M.loca = T.loca$
$M \rightarrow$ U M	$U.loca = M.loca$ $M.loca = U.loca$
$M \rightarrow$ B	$B.loca = M.loca$
$M \rightarrow \epsilon$	
$U \rightarrow \hat{}\{$ $B\}$	$B.loca = sup(U.loca)$
$U \rightarrow \_$ $V\}$	$V.loca = U.loca$
$V \rightarrow \hat{}\{$ $B_1\}\{$ $B_2\}$	$B_1.loca = sup(V.loca)$ $B_2.loca = sup(V.loca)$
$V \rightarrow \{$ $B\}$	$B.loca = sup(V.loca)$
$T \rightarrow id$	$t = mkleaf(id), T.tf = t.tf, T.l = t.l + 0.5 \times t.f$
$T \rightarrow num$	$t = mkleaf(num), T.tf = t.tf, T.l = t.l + 0.5 \times t.f$
$T \rightarrow ($ $B)$	$t = mkleaf("("), B.tf = t.tf, B.l = t.l + t.f \times 0.35$ $t = mkleaf(")"), T.tf = B.tf, T.l = B.l + B.f \times 0.2$
$T \rightarrow \backslash blank$	$t = mkleaf(BLANK), T.tf = t.tf, T.l = t.l + 0.5 \times t.f$
$T \rightarrow \backslash int\{$ $B_1\}\{$ $B_2\}\{$ $B_3\}$	$t = mkleaf(INT), B_1.loca = sub(t.loca)$ $B_2.loca = sup(t.loca)$ $B_3.tf = t.tf, B_3.l = max(B_1.l, B_2.l)$ $T.tf = B_3.tf, T.l = B_3.l + B_3.f \times 0.75$
$T \rightarrow \backslash sum\{$ $B_1\}\{$ $B_2\}\{$ $B_3\}$	$t = mkleaf(SUM), B_1.loca = sub(t.loca)$ $B_2.loca = sup(t.loca)$ $B_3.tf = t.tf, B_3.l = max(B_1.l, B_2.l)$ $T.tf = B_3.tf, T.l = B_3.l + B_3.f \times 0.75$

```
if (ischar(c))
    state = 1;
else if (isdigit(c))
    state = 2;
else if (c == '(' || c == ')' || c == '{' || c == '}' || c == '_' || c == '^')
    state = 3;
else if (c == '\\'&&c1 == 'b')
    state = 4;
else if (c == '\\')
    state = 5;
else if (c == '$')
    state = 6;
else if (c == '\0')
{
    printf("Lexical error!Missing end $.\n");
    return -1;
}
else
{
    LexError(s, i);
    return -1;
}
```

Fig. 5: Codes of error handling in lexical analysis

```
void LexError(char *s, int i)
{
    printf("Lexical error on character:%c.\n", s[i]);
}
```

Fig. 6: Lexical error handling function

error in production. From this, we can see that this program can find and precisely localize errors. The codes are shown in Fig.7.

```
if (ELEMENT[count].type == '_')
{
    printf("U->_V\n");
    match_op('_');
    t=V(loca);
    if (t == -1)
    {
        printf("Syntax error on U->_V  \n");
        return -1;
    }
    return t;
}
```

Fig. 7: Codes of error handling in syntax analysis

3 Experimetal results and analysis

In this part I will show some test samples' results. Full test samples are provided in the class homepage and I will show some here. First, I entered an arbitrary expression:

$$\$h\_\{1\}\backslash int\{\backslash sum\{b\_\{y\}\}\{b \hat{} \{z\}\}\{a\_\{r\}\}\}\{\backslash sum\{c\_\{n\}\}$$
$$\{c \hat{} \{m\}\}\{a \hat{} \{pq\}\}\}\{k\_ \hat{} \{a \hat{} \{m\}\}\{b \hat{} \{n\}\}\}\$$

and the typesetting result is shown in Fig.8. The HTML file is shown in Fig.9.

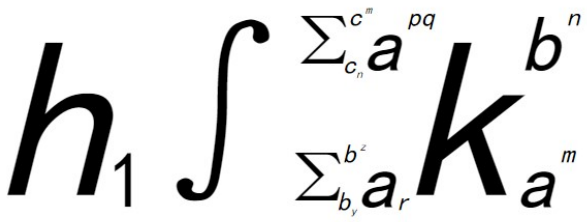


Fig. 8: Typesetting result of an arbitrary input.

Fig. 9: HTML file

Fig.10, 11, 12, 13 show the results of correct test samples (test samples are in the last page). Fig.14, 15, 16 show the incorrect test samples. The former two figures show errors in syntax analysis and the last figure shows an error in lexical analysis.

The expression for Fig.14 is $\$\{a\_\{4\}\} \hat{} \{2\}\$$, the first character of which cannot be "{".

The expression for Fig.15 is $\$a \hat{} \{b \hat{} \{c \hat{} \{2\}d\}\}\$$, which misses a "}".

The expression for Fig.16 is $\$ + a2\_\{6\}\$$, which includes an illegal character "+".

Fig. 10: Results of test samples

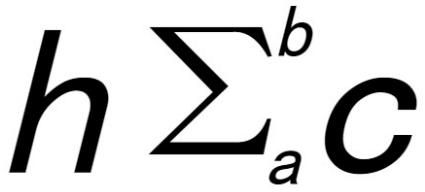


Fig. 11: Results of a test sample

Fig. 12: Results of a test sample

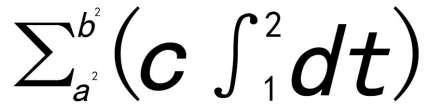


Fig. 13: Results of a test sample

```
syntax:
S->$B$
B->TM
Syntax error on NO.1 element
```

Fig. 14: Result of an incorrect test sample

```
Syntax error on }
```

Fig. 15: Result of an incorrect test sample

```
$+a2_{6}$
Lexical error on character:+
```

Fig. 16: Result of an incorrect test sample

4 Design of typesetting program

This program includes lexical analysis, syntax analysis and syntax-directed translation. syntax analysis and syntax analysis and syntax-directed run simultaneously. The flowchart is shown in Fig.17.

Fig.18 shows the main function of this program. An essential structure of this program is *element*, which is shown in Fig.19. This structure stores each symbol's detailed information after lexical analysis (like $id, number$'s information). The character array *str* stores the specific character/string in the expression and *type* stores the "type"(i.e. $i-id, N-number, ...$). Next, the syntax analyzer can directly analyze the "type" and pinpoint each symbol and its corresponding string in the expression.

4.1 Lexical analysis

The lexical analyzer mainly analyzes the input character stream, which is mainly done by the function $Lex\_analysis()$. The structure $ELEMENT$ records each symbol's information. The main codes are shown in Fig.20, 21. Handling other states like the type S($\backslash sum$) displays the same as above.

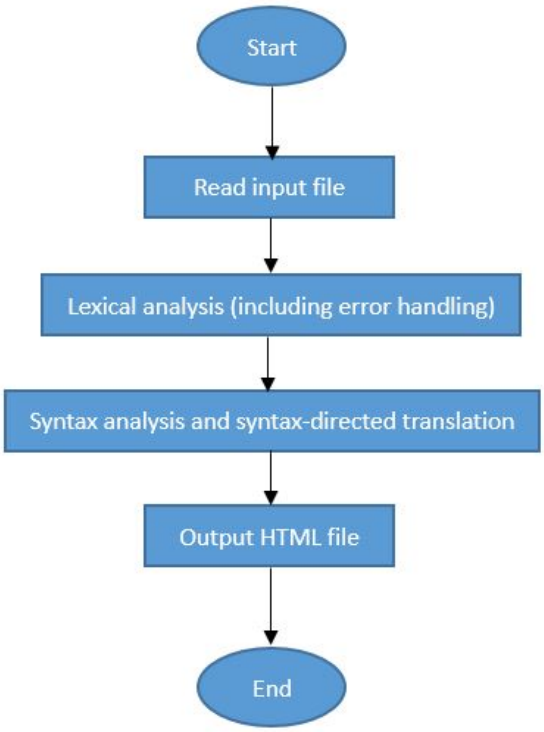


Fig. 17: Flowchart of this program

```
void main()
{
    int loca[3] = { 500,300,200 };   //initial values for 3 parameters
    fp1 = fopen("in.txt", "r"); //open file that needs typesetting
    char *ch_in = new char[100];
    fgets(ch_in, 100, fp1);
    fclose(fp1);
    printf("%s\n", ch_in);
    Lex_analysis(ch_in, ELEMENT);       //Lexical analysis
    fp2 = fopen("out.html", "w");
    //write html file's head
    char *s = "<html>\n<head>\n<META content=\"text/html; charset = gb2312>\"\n</head>\n<body>\n";
    fprintf(fp2, "%s", s);
    S(loca);  //syntax analysis and syntax-directed translation , start from S->$B$
    char *ss = "</body>\n<html>";
    fprintf(fp2, "%s", ss);
    fclose(fp2);
}
```

Fig. 18: Main function of this program

```
typedef struct element{
    char str[10];
    char type;
}ele;
```

Fig. 19: Structure *element* of this program

```
while (flag)
{
    switch (state)   //"state" is the state after read each character
    {
    case 0:
        c = s[i];
        c1 = s[i + 1];
        if (ischar(c))
            state = 1;
        else if (isdigit(c))
            state = 2;
        else if (c == '(' || c == ')' || c == '{' || c == '}' || c == '_' || c == '^')
            state = 3;
        else if (c == '\\'&&c1 == 'b')
            state = 4;
        else if (c == '\\')
            state = 5;
        else if (c == '$')
            state = 6;
        else if (c == '\0')
        {
            printf("Lexical error!Missing end $.\n");
            return -1;
        }
```

Fig. 20: Matching function of terminals

```
case 1:
    strcpy(ELEMENT[j].str, GetId(s, &i));  //GetId reads strings that represents id
    ELEMENT[j].type='i';//i means id
    j++;
    state = 6;
    break;
case 2:
    strcpy(ELEMENT[j].str, GetNumber(s, &i));  //GetNumber reads number strings
    ELEMENT[j].type = 'N'; //N means number
    j++;
    state = 6;
    break;
case 3:
    temp[0] = c; temp[1] = '\0';
    strcpy(ELEMENT[j].str, temp);
    ELEMENT[j].type = c;
    i++; j++;
    state = 0;
    break;
case 4:
    i++;
    for (k = 0; k < 5; k++)
        temp[k] = s[i + k];
    temp[k] = '\0';
```

Fig. 21: Derivation of non-terminals

4.2 Syntax analysis and syntax-directed translation

For LL(1) grammar, there are two main methods employed for syntax analysis, recursive decent parsing and nonrecursive parsing . This program uses the former and writes an analysis for each non-terminal. The analysis table is produced implicitly. However, the non-recursive method needs to produce a predictive analysis table in advance and builds an analysis stack. In syntax-directed translation, the recursive decent method only adds codes to analysis functions, while the non-recursive method has to build a parsing tree. The former method is more concise.

I use an independent matching function to match terminals with symbols, which is shown in Fig.22. The derivation of non-terminals is shown in Fig.23.

In syntax-directed translation, as discussed above, $left, top, font-size$ are stored in array *location*. Calculate the parameters in each step of syntax analysis. The functions *sup*() and *sub*() which calculate the superscript and subscript's coordinates are shown in Fig.24.

The codes that writing the parameters into HTML files are shown in Fig.25(just take several states for example).

4.3 Localization of errors

4.3.1 Lexical analysis

When DFA receives an illegal character, the program prints out this character and its location. Take analyzing $\backslash blank$ as an example, which is shown in Fig.26. Call error handling function if the symbol isn't $\backslash blank$ and then indicate error character and its specific location.

```c
int match_op(char get)          //matching $,(,),{,},_,^
{
    if (get == ELEMENT[count].type)
    {
        count++;
        return 1;
    }
    else   //错误
    {
        printf("Syntax error on %c\n", get);
        return -1;
    }
}
int match_other(char get)   //matching id,num,\blank,\int,\sum
{
    int i = 0;
    if (ELEMENT[count].type == get)
    {
        while (ELEMENT[count].str[i] != '\0')
        {
            i++;
        }
        count++;
        return i;  //the number of characters returned will
                   //be used for calculating coordinates in syntax-directed translation
    }
    else
    {
        printf("Syntax error on %c\n", get);
        return -1;
```

Fig. 22: Matching function of terminals

```c
int S(int *loca)  //S->$B$
{
    int flag;
    int loc;
    flag = match_op('$');
    if (flag == -1) {
        return -1;
    }
    printf("S->$B$\n");
    loc=B(loca);
    flag = match_op('$');
    if (flag == -1)
        return -1;
    else
    {
        printf("Syntax analysis completed\n");
        return 1;
    }
}
int B(int *loca)  //B->TM
{
    int loc;
    printf("B->TM\n");
    loc = T(loca);
    if (loc == -1)
        return -1;
    loc = M(loca);
    return loc;
}
int M(int *loca)   //M->UM | B | null
```

Fig. 23: Derivation of non-terminals

```c
//for superscript
void sup(int * a, int *b)
{
    b[0] = a[0];        //horizontal coordinate
    b[1] = a[1];    //vertical coordinate
    b[2] = a[2] * 0.35;     //font-size
}
//for subscript
void sub(int * a, int *b)
{
    b[0] = a[0];
    b[1] = a[1] + 0.6*a[2];
    b[2] = a[2] * 0.35;
}
```

Fig. 24: Functions *sup*() and *sub*()

```
if (ELEMENT[count].type == 'i')
{
    t=match_other('i');
    printf("T->id\n");
    fprintf(fp2, "\t<div style=\"position: absolute; left:%dpx; top:%dpx;\"><span style=\"font-family:SimHei;font-size:%dpx; f
    return loca[0] + loca[2] * 0.5*t;
}
else if (ELEMENT[count].type == 'N')
{
    t=match_other('N');
    printf("T->num\n");
    fprintf(fp2, "\t<div style=\"position: absolute; left:%dpx; top:%dpx;\"><span style=\"font-family:SimHei;font-size:%dpx; f
    return loca[0] + loca[2] * 0.5*t;
}
else if (ELEMENT[count].type == 'B')
{
    t=match_other('B');
    printf("T->\\blank\n");
    fprintf(fp2, "\t<div style=\"position: absolute; left:%dpx; top:%dpx;\"><span style=\"font-family:SimHei;font-size:%dpx; f
    return loca[0] + loca[2] * 0.5;
}
else if (ELEMENT[count].type == '(')
{
    printf("T->(B)\n");
    match_op('(');
    fprintf(fp2, "\t<div style=\"position: absolute; left:%dpx; top:%dpx;\"><span style=\"font-size:%dpx; font-style:normal;li
    loca[0] = loca[0] + loca[2] * 0.35;
    t = B(loca);
```

Fig. 25: Codes that generating HTML file

```
case 4:
    i++;
    for (k = 0; k < 5; k++)
        temp[k] = s[i + k];
    temp[k] = '\0';
    if (strcmp(temp, "blank") == 0)
    {
        strcpy(ELEMENT[j].str, "\\blank");
        ELEMENT[j].type = 'B';
        j++;
        i += 5;
        state = 0;
        break;
    }
    else
    {
        LexError(s, i);
        return -1;
    }
```

Fig. 26: Error handling in lexical analysis

4.3.2 Syntax analysis

In recursive decent parsing, when there is derivation that doesn't agree with grammar, the program indicates which step has an error with its location. Take $T \rightarrow id|num|(B)|\backslash blank|$ $\backslash int\{B\}\{B\}\{B\}|\backslash sum\{B\}\{B\}\{B\}$ as an example, which is shwon in Fig.27.

```
    loca[0] = loca[0] + loca[2] * 0.2;
    return t + loca[2] * 0.2;
}
//all legal derivations have been analyzed , other syntax errors are handled here
else
{
    printf("Syntax error at NO.%d element     on R ->id | num | \blank | (B) | \sum{B}{B}{B} | \int{B}{B}{B} \n", count++);
    return -1;
}
```

Fig. 27: Error handling in syntax analysis

5 Summary

I met many problems when conducting this project. At the beginning of the syntax analysis, I didn't realize the significance of the ambiguity problem and there were program errors in the analysis of legal statements. From this point forward I started to think profoundly and discover problems. It indicates the significance of theoretical analysis. I need to analyze based on the characteristics and regulations of specific grammar and then start programming. At the step of semantic analysis, I found the overlapping problems of some characters. It required me to adjust my expression, which focuses on calculating location parameters.

The programming work in this projects in not challenging, but the difficulty lies in the theoretical lexical analysis, syntax analysis and syntax-direction translation for given text, and modifying grammar according to a specific analyzing method. During the project, I gained a clearer understanding of lexical, syntax and semantic analysis, as well gained a more nuanced understanding in LL and LR analysis.

References

[1] Yiyun Chen and Yu Zhang. *Principles of Compilers*. Higher Education Press, 2008.

Instructions of typesetting system

1. Character set

Lowercase: a, b, …, z A, B, …, Z
Number: 0, 1, …, 9
Special symbol: \, (,), {, }, _, ^, $
Separator: blank, line feed

2. Definition

Identifier: It begins with a letter, consists of letters and numbers and displays as italic.
Number: unsigned integer displays as normal font

3. Language rules

Sentences: $α$
superscript and subscript:
Superscript: ^{α}; subscript: _{α}. When superscript and subscript coexist, use _^{α}{β}, among which α is subscript, β is superscript,
bracket：Only use ()
blank: \blank
Large-scale operation:
Integration: \int{α}{β}{γ}
Accumulation: \sum{α}{β}{γ}, α is lower limit, β is upper limit, γ is content of operation

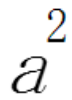
4. Basic grammar

(There may be some faults. You can change it if necessary)
S→B
B→BB
B→B_^{B}{B}
B→B^{B}
B→B_{B}
B→\int{B}{B}{B}
B→\sum{B}{B}{B}
B→id | num | \blank | (B)

5. input and output

5.1 input: text file, one text file only includes one expression
File name of sample file: sample??.txt
File name of test file: test??.txt
5.2 Output: hypertext markup language file.
File name of sample file: sample??.html
File name of test file: test??.html

5.3 Samples
(1) a^{2}



HTML codes:
<html>
<head>
<META content="text/html; charset=gb2312">
</head>
<body>
<span style="font -size:50px;
font-style:oblique;
line-height:100%;">a
<span style="font -size:30px;
font-style:normal;
line-height:100%;">2
</body>
</html>
(2) $a_^{c2}{b}$

HTML codes:
<html>
<head>
<META content="text/html; charset=gb2312">
</head>
<body>
<span style="font -size:50px;
font-style:oblique;
line-height:100%;">a
<span style="font -size:30px;
font-style:oblique;
line-height:100%;">c2
<span style="font -size:30px;
font-style:oblique;
line-height:100%;">b
</body>
</html>

Test Samples

Correct Test samples:

1. a^{2}

2. $a_^{c2}{b}$

3. a_{2}

4. $h\sum{a}{b}{c}$

5. $(thisIS512)$

6. $\sum{a^{2}}{b^{2}}{(c\int{1}{2}{dt})}$

Incorrect Test Samples:

1. ${a_{4}}^{2}$

2. $a^{b^{c^{2}d}$

3. $+a2_{6}$