ITP 20002-03 Discrete Mathematics, Fall 2021

Homework 3

Your name : Seokjun Kim(김석준)

Email address : 21600081@handong.edu

Generating Languages from Grammars

1. Introduction

Language is a set of strings that satisfy certain rules called grammar. This program is a program that receives the grammar of the language as input and outputs all members corresponding to the language. The grammar configured by the derivation rules starts with a starting symbol 'S' and is rewritten, that is replaced. The key point is that a recursion occurs until the symbol in the right-hand side of the grammar is continuously replaced and expressed only as a concrete string without a symbol, that is until it is expressed as a member of the language. What should be noted here is that the same concrete string can be produced by different derivation rules depending on the grammar. Since the language is a set, if some concrete string come out several times, only one element can be a member of the language. Therefore, this should be considered well in this program. Another thing to consider is that the symbol can be rewritten infinitely in the process of being replaced in grammar. So, if we do not specify the maximum length of the language, in most grammar, the number of language members will be infinitely many, and the program will not end. So, we will receive grammar file and a maximum length of the language through the command line interface.

2. Approach

Through the CLI, receive grammar and maximum length, and make an array to store grammar line by line. Initially, we need to find the right-hand side of the starting symbol 'S' and find the symbol of RHS to be rewritten. Then we use this grammar and a symbol to recurs and if this symbol is found in grammar's left-hand side, we will replace this symbol with its corresponding derivation rule and store its string. If this string is saved and put in the recursion part, the rewritten can occur well. In this process, if the length of the concrete string, including symbol, exceeds the length we received as an input, we can end the recursion for the grammar to prevent meaningless repetition. If there is no symbol to be placed in the string, it means that it is a concrete string, which means that it is a member of language. Here, if this string is output, all members of the language may be output. However, when output in this process, repeated members may be printed. Therefore, using the linked list data structure, whenever there is no symbol to be replaced, a new list is created and connected. At this time, if we connect the new list to the last list when it does not overlap while checking whether the new member to be added from the beginning is the same as the existing member, there will be no redundant member when all the replacements are finished.

```c
int main(int argc, char *argv[])
{
    char *file = argv[1];
    int length = atoi(argv[2]); // CLI =  ./langdump ??.txt length
    FILE *fp;
    char buffer[MAX_LENGTH]; // to save grammar in array
    char grammar[MAX_LENGTH][MAX_LENGTH];
    int grammar_size = 0; // for the loop in recursive function

    fp = fopen(file, "r");

    while (fgets(buffer, MAX_LENGTH, fp))
    {
        strcpy(grammar[grammar_size], buffer);
        grammar_size++;
    }
    find_member(grammar, grammar_size, length);

    printf("================\n");
    printList(resultlist);
    free(resultlist);
    return 0;
}
```

In the main function, use grammar and length to call the find_member function that stores the members belonging to the language in the linked list.

```c
void find_member(char (*grammar)[MAX_LENGTH], int grammar_size, int length)
{
    resultlist = malloc(sizeof(resultList));
    resultlist->next = NULL;

    char symbol[MAX_LENGTH];
    erase_newline(grammar, grammar_size);
    for (int i = 0; i < grammar_size; i++)
    {
        if (grammar[i][0] == 'S' && grammar[i][1] == ' ')
            strcpy(symbol, strstr(grammar[i], "::= ") + 4);
    }

    if (symbol == NULL)
    {
        printf("given input is invalid.\n");
        return;
    }
    char result[sizeof(symbol)];
    strcpy(result, symbol);

    replace(grammar, grammar_size, result, symbol, length, resultlist);
}
```

In find_member function, we store the right-hand side of the left-hand side of a line with starting symbol 'S' in grammar as symbol, and input it as an argument in the replace function which is recursive function.

```c
void replace(char (*grammar)[MAX_LENGTH], int grammar_size, char *result, char *symbol, int length, resultList *resultlist)
{
    if (check_length(result) > length)  // check length
        return;
    printf("================\n");
    printf("String will be rewritten: %s\n", result);
    for (int i = 0; i < grammar_size; i++)
    {
        printf("= = = = = = = = =\n");
        printf("symbol : %s\n", symbol);
        printf("grammar : %s\n", grammar[i]);
        if (strncmp(grammar[i], symbol, strlen(symbol)) == 0)  // check symbol of grammar
        {
            char *rhs;
            rhs = strstr(grammar[i], "::= ") + 4;
            char new_result[MAX_LENGTH];
            int new_length = 0;
            char *symbolPtr = find_symbol(result);

            if (symbolPtr == NULL)
            {
                strcpy(new_result, rhs);
            }
            else
```

In replace function, check the length of the result string that we need to replace first. The check_length function is a function that returns only the length of a meaningful string except for the quotation mark and space. Then, check the left-hand side for each line of grammar to see if it is the same symbol we have to replace.

```c
            {
                for (int i = 0; i < strlen(result) - strlen(symbolPtr); i++)
                {
                    new_result[new_length] = result[i];
                    new_length++;
                }
                new_result[new_length] = '\0';
                strcat(new_result, rhs); // rhs = 교체될 문법
                symbolPtr += strlen(symbol) + 1;
                strcat(new_result, " ");
                strcat(new_result, symbolPtr);
            }

            printf("result : %s\n", result);
            printf("new_result : %s\n", new_result);
```

Replace the symbol in new_result and put a new string for the recursion part again. (e.g. E → “1” E “1”)

```c
            char temp_result[MAX_LENGTH];
            if (find_symbol(new_result) == NULL)
            {
                int temp_length = 0;
                int isConcrete = 0;
                for (int p = 0; p <= strlen(new_result); p++)
                {
                    if (new_result[p] == '\"')
                    {
                        isConcrete++;
                        continue;
                    }
                    else if (new_result[p] == ' ' && isConcrete % 2 == 0)
                    {
                        continue;
                    }
                    else if (new_result[p] == ' ' && isConcrete % 2 != 0)
                    {
                        temp_result[temp_length] = new_result[p];
                        temp_length++;
                        continue;
                    }
                    else if (new_result[p] == '\0')
                    {
                        temp_result[temp_length] = '\0';
                        break;
                    }
                    else
                    {
                        temp_result[temp_length] = new_result[p];
                        temp_length++;
                    }
                }

                printf("result in list : %s\n", temp_result);
                if (strlen(temp_result) <= length)
                {
                    addList(resultlist, temp_result);
                }
                else
                    break;
            }
```

The find_symbol function is a function that returns the pointer at the corresponding memory when the string finds a symbol other than the concrete string. Using this, if there is no symbol to replace, only the concrete string is added to the temp_result and it is linked to the list.

```c
            else
            {
                symbolPtr = find_symbol(new_result);
                char newSymbol[16];
                int newSymbol_length = 0;
                while (symbolPtr[0] != ' ')
                {
                    newSymbol[newSymbol_length] = symbolPtr[0];
                    symbolPtr++;
                    newSymbol_length++;
                }
                newSymbol[newSymbol_length] = '\0';
                printf("newSymbol : %s\n", newSymbol);
                replace(grammar, grammar_size, new_result, newSymbol, length, resultlist);
            }
        }
    }
}
```

This part is a recursive part. Since there is a symbol to be replaced, the symbol is changed to a new symbol, and it is used an in argument for recursive function with a string rewritten by the derivation rule.

```c
void addList(resultList *resultlist, char *newMember)
{
    if (resultlist->next == NULL)
    {
        resultList *newList = malloc(sizeof(resultList));
        strcpy(newList->member, newMember);
        newList->next = NULL;

        resultlist->next = newList;
    }
    else
    {
        resultList *curr = resultlist;
        while (curr->next != NULL)
        {
            if (strcmp(curr->member, newMember) == 0)
                return;
            curr = curr->next;
        }

        if (strcmp(curr->member, newMember) == 0)
            return;
        resultList *newList = malloc(sizeof(resultList));
        strcpy(newList->member, newMember);
        newList->next = NULL;

        curr->next = newList;
    }

    return;
}
```

Through this addList function, redundancy can be reduced.

3. Evaluation

In order to check if the program works well, I wrote a grammar of bit strings only contains 1. For deliberate repetition, 1 was added to the front and back of the symbol so that the same concrete string could come out with another derivation rule.

(e.g.) S → E → E "1" → "1" "1”

S → E → "1" E → "1" "1"

```
test.txt
S ::= E
E ::= "1"
E ::=  E "1"
E ::= "1" E
```

```
================
1
11
111
1111
11111
================
5
```

The result of ./langdump test.txt 5 is as above, and it can be seen that it is output well uniquely without exceeding the maximum length. In fact, grammar as above is a grammar that can be appropriate grammar even if the last line is deleted, but this grammar is also a grammar that represents what bit strings only contains 1, so it must work properly.

Now let’s check out the three problems.

1) grammar of binary palindrome strings

```
palindrome.txt
S ::= E
E ::= "0"
E ::= "1"
E ::= "0" "0"
E ::= "1" "1"
E ::= "0" E "0"
E ::= "1" E "1"
```

```
========
0
1
00
11
000
010
0000
0110
00000
00100
```

```
000000
001100
01010
01110
010010
011110
101
111
1001
```

```
1111
10001
10101
100001
101101
11011
11111
110011
111111
========
28
```

The number of binary palindromes having length n is 2^n when n is odd and 2^{n-1} when n is even. Therefore, the number of binary palindromes up to length 6 is 2+2+4+4+8+8=28. If you check the above results, you can see that all of them are binary palindrome and there is no repetition. In fact, the grammar of binary palindrome is a grammar in which there cannot be repetition.

2) grammar of well-balanced and properly nested parentheses, curly brackets, square brackets.

```
wellfix.txt
S ::= SB
SB ::= SB SB
SB ::= "[" SB "]"
SB ::= "[" "]"
SB ::= CB
CB ::= "{" CB "}"
CB ::= "{" "}"
CB ::= P
P ::= "(" P ")"
P ::= "(" ")"
```

```
()[(())]
()[()]
()[]
(){{{{}}}}
(){{{()}}}
(){{{}}}
(){{(())}}
(){{()}}
(){{}}
(){((()))}
(){(())}
(){()}
(){}
()(((())))
()((()))
()(())
()()
[][][][]
[][][]{}]
[][][]()]
[][]{}[]]
[][]{}{}]
[][]{}()]
```

```
{{{(())}}}
{{{()}}}
{{{}}}
{{((()))}}
{{(())}}
{{()}}
{{}}
{((((()))))}
{((()))}
{(())}
{()}
{}
(((())))) 
((()))
(())
()
================
3544
```

We confirmed that there is no repetition through the previous example. To see if all the printed members of this language are suitable, check that there is a left parenthesis in the leftmost side and right parenthesis in the rightmost side.

3) grammar of postfix arithmetic expressions

```
PA2 > postfix.txt
S ::= E
E ::= N N O
E ::= N E O
E ::= E N O
N ::= "1 "
N ::= "2 "
N ::= "3 "
N ::= "4 "
O ::= "+ "
O ::= "- "
O ::= "* "
O ::= "/ "
```

```
4 4 3 * +
4 4 3 * -
4 4 3 * *
4 4 3 * /
4 4 3 / +
4 4 3 / -
4 4 3 / *
4 4 3 / /
4 4 4 + +
4 4 4 + -
4 4 4 + *
4 4 4 + /
4 4 4 - +
```

```
4 4 / 1 +
4 4 / 1 -
4 4 / 1 *
4 4 / 1 /
4 4 / 2 +
4 4 / 2 -
4 4 / 2 *
4 4 / 2 /
4 4 / 3 +
4 4 / 3 -
4 4 / 3 *
4 4 / 3 /
4 4 / 4 +
4 4 / 4 -
4 4 / 4 *
4 4 / 4 /
================
2112
```

Postfix arithmetic expression consists of two numbers and one operator. In addition, since the result value of the postfix arithmetic expression is also a number, the number can be another arithmetic expression. Therefore, postfix AE up to length 10 has a maximum of 3 numbers and 2 operators. The number ranges from 1 to 4, and there are four operators. Let n be numbers, and o be operators, then possible cases are $n\,n\,o$, $n\,n\,n\,o\,o$, $n\,n\,o\,n\,o$ so the total number of language members up to length 10 is $4^3 + 4^5 + 4^5 = 2112$.

From these three examples, this program is well outputting members up to a given length for a given grammar.

4. Discussion

In the process of writing this program, it was found that if the grammar became complicated or if the maximum length was set long, it would take too long. Perhaps it is not good way to put in the linked list because it uses dynamic allocation and adds a list while checking from beginning to end. I compared the time by organizing a program that does not use dynamic allocation. However, there was no time difference between programs with dynamic allocation and programs without. I do not know the exact reason, but I think it is because I set the same MAX_LENGTH for making a string array.

The ways, which make the program efficient, and make the grammar less to avoid repetition, would both be ways to save time. However, for the developer, I think the former method is much more important because the user should be able to save time no matter how the user gives some grammar. Maybe it is because I'm not used to C yet, but I thought it was not easy to deal with strings in C. However, I think it is an advantage to be able to control the characters in the string through a pointer, set the null character well to handle it in the program to think of it as a variable other than a set string.

5. Conclusion

What I thought was important in creating this program was that first, what were language and grammar, and the strings created by the derivation rule could not be repeated because the language is a set. There are still many shortcomings in this program, but I think I have created a program that suits the purpose to some extent, and through this program, I think I have a better understanding of language, grammar, and recursion. Also, in the process of programming, I found that it took a very short time to call a function, even though there were countless repetition for the recursion. The printing process took up much more time. In addition to a program that outputs all members of a language, I want to create a program that can check whether they are members of a specific language.