Term Project (2024/10)

In this term project, you have to construct a so called semi-C interpreter. The interpreter prints out function names and the values of actual parameters whenever the functions are called during a program execution. The construction of the interpreter largely consists of two parts. In the first part, lexical and syntax analysis are conducted. Generate **a flow graph** representing the source program and create a symbol table to hold the variable or function values. The changes of variable and function values during a program execution must be registered and the corresponding fields should be updated in the symbol table.

In the second part, traverse the flow graph constructed in the first part and print out function names and the values of actual parameters whenever the functions are encountered during the running of the program. When a value is assigned to a variable by an **assignment statement**, find the variable in the symbol table and store the value in the value field of the variable. For the computation of an **expression**, you may need to refer to the values stored in the symbol table. Special care is necessary for the treatment of **if** statement and **loop** structure. Note that the control flow is decided by the evaluation of the condition statements. To handle **function call** properly, stack manipulation is indispensable. Think over how you can handle **recursive** procedures without stack manipulation.

Your interpreter has to support type checking and memory management. To support the memory management in your interpreter, allocate enough heap space at the starting point of the program. At the beginning, the whole space is used as a free space. When there are requests for memory allocations from the program, some spaces are allocated and the rest remained spaces are used as the free space. If there is no more free space to allocate, you may need to collect garbage in the heap space. There are many different memory allocation strategies you can use. Select one of the strategies that you think is easiest to implement and monitor what happens in the heap space. Extra points will be given if your interpreter can generate assembly code. You may use any kind of assembly code for the code generation. (cc -S sample.c or gcc -S sample.c)

In the below, we show an example input program and an expected result that your interpreter has to generate. A sample graph that has to be constructed by your interpreter will be explained in the class.

(Note) You may define your own sub-C grammar that can specify and translate at least the below example program.

Example input program

```
avg(int count, int *value){
    int total;
    sum = 0;
    for (i =1; i < count; i++ ) {
        total = total +  value[i];
    }
    return(total/count);
}

int main(void){
    int studentNumber, count, i, sum;
    int mark[4];
    float average;

    count = 4;
    sum = 0;
    for (i =1; i < count; i++ ) {
        mark[i] = i * 30
        sum = sum +  mark[i];
        average = avg(i+ 1, mark);
        if (average > 40) {
                printf("%f", average);
         }
    }
```

Expected output

```
        avg(1, [30, 0, 0])
        avg(2, [30, 60, 0])
        return(45) printf("%f", 45)
        avg(3, [30,60,90])
        return(60) printf("%f", 60)
```

Due date: Dec. 13th, 2024
Term Project Progress Presentation: Nov. 26th, 2024