CS348-Assignment 6

Sweeya Reddy 200101079
Vatsal Gupta 200101105

README

25th April 2023

ZIP FILE CONTENTS

1. A6_40.c
2. A6_40.l
3. A6_40.y
4. A6_40_target_translator.cpp
5. A6_40_translator.cpp
6. A6_40_translator.h
7. Makefile
8. bin(dir)
9. myl.h
10. test-inputs(dir)
    a. A6_40_test1.c
    b. A6_40_test2.c
    c. A6_40_test3.c
    d. A6_40_test4.c
    e. A6_40_test5.c
    f. A6_40_test6.c
11. test-outputs(dir)
    a. A6_40_1.o
    b. A6_40_1.s
    c. A6_40_2.o
    d. A6_40_2.s
    e. A6_40_3.o
    f. A6_40_3.s
    g. A6_40_4.o
    h. A6_40_4.s
    i. A6_40_5.o
    j. A6_40_5.s

k. A6_40_6.o
l. A6_40_6.s
m. A6_40_quads1.out
n. A6_40_quads2.out
o. A6_40_quads3.out
p. A6_40_quads4.out
q. A6_40_quads5.out
r. A6_40_quads6.out

12. Readme.pdf(here)

** Note some files are created on running the make command these are auto-generated intermediaries crucial to the running of the program, they are given in the submission and can be remade by using the make commands.*

PROGRAMMING LANGUAGE USED (upon UBUNTU 20.04 LTS)

nanoC, C,C++(with C++17 and g++11 compiler standards.), Lex/Flex (lexer), (parsing)Bison

Key Points/Pre-requisites

- Please make sure the terminal is run in the same directory (cwd) as the listed files above
- The program will first create A6_40.tab.c and A6_40.tab.h files from the parser A6_40.y by using the bison shell commands.
- The program will create a .c file - lex.yy.c from A6_40.l, lex file using the A6_40_translator.h & A6_40.tab.h file as index lister and headers and subsequently the required parser files like y.tab.h and auto generate token numbers and then use them to parse the input later.
- 4 object files are created and subsequently used for finally generating the executable required as follows
    - A6_40.tab.o
    - A6_40_translator.o
    - A6_40_target_translator.o
    - Lex.yy.o
- These finally produce our nanoC executable using the g++ command:
  `g++ lex.yy.o A6_40.tab.o A6_40_translator.o A6_40_target_translator.o -lfl -o nanoC`
- **The output is generated after creating & using an internal symbol table which is implicitly stored as a map in the c++ file and code. The debug print statements as in**

previous assignments may be used to see the codes generated from the tokens returned and the productions applied.

Commands to enter and expected output:

```
make test
```

- Simply type make test into the command line and then the test-outputs directory is filled up with 3 files for each test case input as given in test-inputs. This consists of ,for the x^{th} testcase:
  - A6_40_<x>.s : an assembly file which generates the .o
  - A6_40_<x>.o : an object file which generates the final executable
  - A6_40_quads<x>.out : An output depicting the quads and the symbol table as in Assignment-5
- This finally generates and places the executable named test<x> into the bin folder
- To run the executable, type the below command (assuming you are still in the directory of the assignment having the bin folder within it):

```
./bin/test<x>
Example: ./bin/test1
```

- **BONUS:** You may also use some debug statements as in Assignment-4 to print colored output onto the terminal, re-use Assignment 4 code in such a scenario.
- You can also retest the output by running make clean first which removes the optional files and then run make again which recreates them proving the sample output is indeed correct. Make clean will essentially employ the following function:

  `“rm -f lex.yy.c *.tab.c *.tab.h *.output *.o *.s *.a *.out *.gch nanoC test-outputs/* bin/*”`

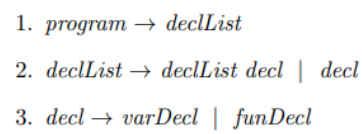
```
make clean
```

Note:

1. **BOUNS:** We have shown the output for all the 6 examples as given in the assignment they can be seen in the quad files and their executables can be run through the bin folder.
2. Our test files are aimed and targeted to test the following things (the codes mainly use the nanoC files already given in assignment-3)
   - The first is the code for bubble sort which can test array access, input into array with appropriate offsets,print functionality for string as well as int and temporary local variable creation.

- Second is the externalized swap function which checks for correct pointer access, calling of an external subroutine/function and reference variables by swapping
- Third is factorial function by iteration which checks for repeated same integer access and modification and printing of the updated value.
- Fourth is checking max of n numbers through array checks for multi-array insertion and retrieval with appropriate comparison and new local variable storage
- Fifth checks for pointer access through addresses and also checks for dereferencing operation
- Sixth simply checks if the same memory location can be accessed repeatedly with internal modifications and whether the updations in the symbol table for memory locale access is done correctly.

3. The lexer follows a certain priority order as given in assignment 3, and the same lexer as submitted earlier has been used with minor modification
4. We have modified some grammar rules, to make the production rules correct and perform right. We have taken inspiration from several sources like the following segment of grammar on the right (A Grammar for the C-Programming Language (Version S21) March 23, 2021 1 Introduction)



5. The changes are done for the translation_unit segment and the function_definition segment all the others are retained, and there are some intermediates defined as interim and interim_decl which are necessary from a code point of view to apply appropriate actions to the IDENTIFIER token when it is received before applying subsequent function_definition production and similarly interim_decl applies appropriate pre-processing to declarator.

Cases Handled:

I. Quads are generated correctly for each opcode token needed and the local as well as function and temporary symbol table creations are handled exactly as in assignment-5 and in the right fashion.
II. Local arrays work correctly, global arrays are easy to handle but omitted assymbol tables may be ambiguous at a later stage, and we were instructed to omit some functionality in assignment 6. Their handling fashion can be seen in Assignment-5 to generate the correct output , they are skipped as a design choice in Assignment-6.
III. Local arrays are accessed for read and write.
IV. Pointer access for read and write and de-referencing is working in directly tying up to the symbol table and is working right.

V. sub-routine s and external function calling is working flawlessly by an extensive use of goto statements and backpatching
VI. If-else conditional and for loop blocks are working correctly for the scope as mentioned in the assignment.
VII. All the test-cases and what they test for is working correctly and the local data structures created in the code mimic the printed symbol table and they are used for the TAC generation as required. The working code and comments explain each data structure, member and functionality and can be reviewed. Here, quad arrays and quads are used as additional data structure and final asm output is inspired form the gnu C++ std compiler to write asm__volatile blocks for various functionality as needed.

Design of Memory Binding is explained in the code comment and TAC generation along with appropriate quad bindings and symbol tables is explained in the code.

Handling of Activation Records & Handling of Static Memory & Binding Design of Code Translation is done correctly as can be seen from the final executable and assembly file being generated

Handling of Prologue & Epilogue Handling & Function Body Design of Target Code Management is done flawlessly as can be seen from the correct temporary quad table being generated and the correct goto bindings for teh subroutines in the assembly file, the same is explained in the code as well.

Integration of translated codes into an assembly file is done correctly by using an fstream output to directly write the chunks of asm code into the asm file.

Appendix

Output Screenshots:

1. **Test case 1:**

```
Input array size:
4
Input array elements:
Input array:
Input next element
34
Input next element
12
Input next element
65
Input next element
43
34 12 65 43
12 34 43 65
```

2. **Test case 2:**

```
Input first element:
54
Input second element:
32
Before swap:
x = 54 y = 32
After swap:
x = 32 y = 54
```

3. **Test case 3:**

```
Input number to find factorial:
6
6! = 720
```

4. **Test case 4:**

```
Input number of elements:
4
Input next element
45
Input next element
32
Input next element
1
Input next element
104
Max of: 45, 32, 1, 104: = 104
```

5. **Test case 5:**

```
9
20
```

6. **Test case 6:**

```
Printing array elements:
0 1 2 3 4
```