CS3300: Compiler Design **Odd 2018**

Assignment #3

Assembly Code Generation (20 marks) Deadline: 30/09/2018, 11:55PM

What you should do?

You are required to,
(i) extend the Abstract Tree generation of Assignment #2 to include print statement, and
(ii) generate assembly code for the language used in Assignment #2 (including print statement) with the help of generated Abstract Syntax Tree from (i). The extended grammar which includes print statement from Assignment #2 is shown in Appendix A.

For which basic constructs assembly code should be generated?

We will be using the grammar given in the Assignment #2 with extension to print statement (Appendix A) and we will restrict assembly generation only to few constructs. Our test-cases (.c file) will contain only the below basic constructs for which you are required to generate the assembly code,

- Variable declaration statements - You need to handle only integer type declarations (excluding pointers in that).
- Assignment statements - Includes all but excluding Array and Function assignment statements.
- Function calls - You need to handle only 'printf' function call. User-defined function and others are excluded.
- If statement - Includes both 'If then' and 'If then else'. The nesting of If statement is excluded.
- While statement - Excluding nesting of 'While' statement , 'break' and 'continue' constructs. Also, no combination of If and While statements together (i.e.) no If stmt inside While and vice-versa.

Bonus (2 marks): You should handle below constructs to get the bonus marks,

- Functions - You should handle user-defined functions and return statements (1 mark).
- Handling Array and Function assignment statements (1 mark).

Input

The input will be a program (.c file) which contains statements according to the above basic construct(s) description.
Sample execution format:

```
$ ./a.out < test_case1.c
```

Output

You should emit the assembly code for the corresponding input file (here test_case1.c), which we will be captured to a .s file like,

```
$ ./a.out < test_case1.c > tc1.s
```

Submission (in moodle)

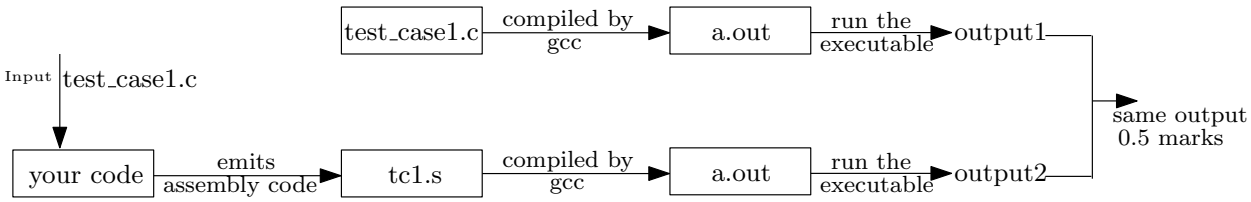
Submit a tar.gz file with filename as ⟨ROLLNO⟩.tar.gz (eg.CS12B043.tar.gz) containing the following structure:

- CS12B043 ⟨directory⟩
  - *.l
  - *.y
  - Makefile

The Makefile should run **lex**, **yacc**, compile the generated code and generate an executable **a.out** file.

How we will evaluate your code?

- We will evaluate your code against 44 test-cases. 8 public test-cases, 32 hidden test-cases, and last 4 test-cases for Bonus marks.
- Each test-case carries .5 marks and total marks is 20. Bonus is 2 mark (last 4 test-cases).
- Below figure shows how your code is evaluated,



- The 44 test-cases will contains a mix-up of very simple, simple, less complex and complex test-cases. We will look only at the final printed output not your generated assembly code. So, you must generate the assembly code for 'printf' function call to pass all test-cases.
- As the assembly code varies across the machines, we will evaluate your code on any one of the following DCF Machines listed (IP address) below,
  - 10.6.15.136; 10.6.15.137; 10.6.15.138; 10.6.15.139; 10.6.15.140.
- So, please make sure your emitted assembly code compiles, executes and generates same output on any one of the listed DCF machines.

Sample Test Case

- **testcase_1.c**
  - **Input**

```
int main() {
        int a;
        a = 6;
        printf("%d\n", a);
        return 0;
}
```

- **Your emitted assembly code stored in tc1.s file**

```
.LC0:
        .string "%d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $6, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

- **evaluation**

```
$ gcc testcase_1.c
$ ./a.out
6
$ gcc tc1.s
$./a.out
6
```

- here both output result matches and will be given 0.5 marks.

What will we upload in the moodle?

- This file (Assignment3.pdf), 8 + 2 public test-cases (last two are bonus test-cases) and evaluation script.
- Code which generates Abstract syntax tree of Assignment #2. You can extend this and generate assembly code or use your own code.
- All 44 test-cases (after the deadline).

Some stuff(s) to know

- You can generate assembly code using gcc. how? $gcc -S testcase_1.c
- A tool named MOSS, which can compare multiple codes and identifies which parts of the code are similar (copied).

FAQs

- You are allowed to use both C and C++ languages for coding.
- "Can I use STL template, xx library calls, and others" - you are allowed to use if it works on any one of the listed DCF machines.
- Only accepted printf stmt used in test cases as per the grammar is “printf("%d\n", identifier);”
- The marks uploaded in moodle will be out of 20.
- Emit your code only using ⟨STDIO⟩, not using ⟨STDERR⟩.
- Remove all your debugging statements.

APPENDIX A: Extended Grammer of Assignment #2

program → decl_list
decl_list → decl_list decl | decl
decl → var_decl | func_decl
var_decl → type_spec identifier ";"
| type_spec identifier "," var_decl
| type_spec identifier "[" integerLit "]" ";"
| type_spec identifier "[" integerLit "]" "," var_decl
type_spec → "void" | "int" | "float"
| "void" "*" | "int" "*" | "float" "*"
fun_decl → type_spec identifier "(" params ")" compound_stmt
params → param_list | ϵ
param_list → param_list "," param | param
param → type_spec identifier | type_spec identifier "[" "]"
stmt_list → stmt_list stmt | stmt
stmt → assign_stmt | compound_stmt | if_stmt | while_stmt |**print_stmt**
| return_stmt | break_stmt | continue_stmt
expr_stmt → expr ";"
while_stmt → "while" "(" expr ")" stmt
print_stmt → **"printf("format_specifier"," identifier");"**
compound_stmt → "{" local_decls stmt_list "}"
local_decls → local_decls local_decl | ϵ
local_decl → type_spec identifier ";"
| type_spec identifier "[" expr "]" ";"
if_stmt → "if" "(" expr ")" stmt
| "if" "(" expr ")" stmt "else" stmt
return_stmt → "return" ";" | "return" expr ";"
break_stmt → "break" ";"
continue_stmt → "continue" ";"
assign_stmt → identifier "=" expr | identifier "[" expr "]" "=" expr
expr → Pexpr "||" Pexpr
→ Pexpr "==" Pexpr | Pexpr "!=" Pexpr
→ Pexpr "<=" Pexpr | Pexpr "<" Pexpr | Pexpr ">=" Pexpr | Pexpr ">" Pexpr
→ Pexpr "&&" Pexpr
→ Pexpr "+" Pexpr | Pexpr "-" Pexpr
→ Pexpr "*" Pexpr | Pexpr "/" Pexpr | Pexpr "%" Pexpr
→ "!" Pexpr | "-" Pexpr | "+" Pexpr | "*" Pexp | "&" Pexp
→ Pexpr
→ identifier "(" args ")"
→ identifier "[" expr "]"
Pexpr → integerLit | floatLit | identifier | "(" expr ")"
integerLit → <INTEGER_LITERAL>
floatLit → <FLOAT_LITERAL>
identifier → <IDENTIFIER>
format_specifier → **"“%d\n”"**
arg_list → arg_list "," expr | expr
args → arg_list | ϵ