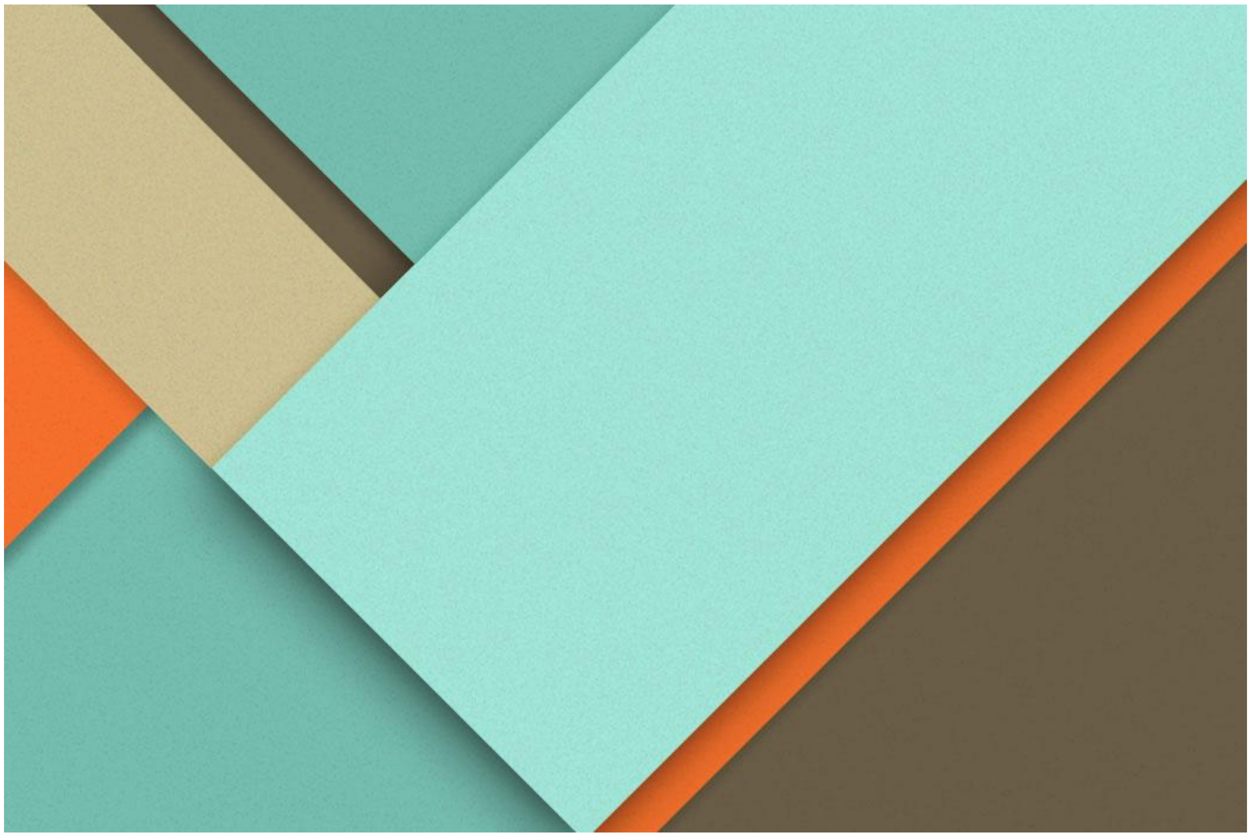


A6 - Target Code Generator for nanoC

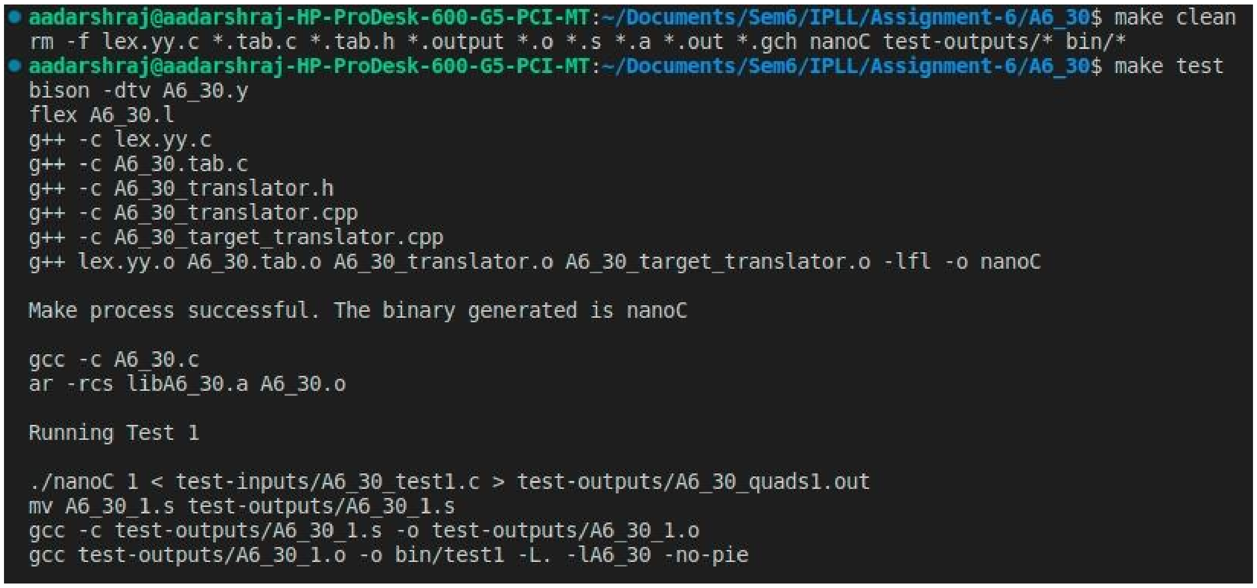
24.04.2023

Group 30

Aadarshraj Sah (200101001)

Mahek Vora (200101062)

Command to run



```
aadarshraj@aadarshraj-HP-ProDesk-600-G5-PCI-MT:~/Documents/Sem6/IPLL/Assignment-6/A6_30$ make clean
rm -f lex.yy.c *.tab.c *.tab.h *.output *.o *.s *.a *.out *.gch nanoC test-outputs/* bin/*
aadarshraj@aadarshraj-HP-ProDesk-600-G5-PCI-MT:~/Documents/Sem6/IPLL/Assignment-6/A6_30$ make test
bison -dtv A6_30.y
flex A6_30.l
g++ -c lex.yy.c
g++ -c A6_30.tab.c
g++ -c A6_30_translator.h
g++ -c A6_30_translator.cpp
g++ -c A6_30_target_translator.cpp
g++ lex.yy.o A6_30.tab.o A6_30_translator.o A6_30_target_translator.o -lfl -o nanoC

Make process successful. The binary generated is nanoC

gcc -c A6_30.c
ar -rcs libA6_30.a A6_30.o

Running Test 1

./nanoC 1 < test-inputs/A6_30_test1.c > test-outputs/A6_30_quads1.out
mv A6_30_1.s test-outputs/A6_30_1.s
gcc -c test-outputs/A6_30_1.s -o test-outputs/A6_30_1.o
gcc test-outputs/A6_30_1.o -o bin/test1 -L. -lA6_30 -no-pie
```

`make clean`

This removes all the output files:

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

`make test`

```
bison -dtv A6_30.y

flex A6_30.l

g++ -c lex.yy.c

g++ -c A6_30.tab.c

g++ -c A6_30_translator.h

g++ -c A6_30_translator.cpp

g++ -c A6_30_target_translator.cpp

g++ lex.yy.o A6_30.tab.o A6_30_translator.o
A6_30_target_translator.o -lfl -o nanoC
```

```
gcc -c A6_30.c

ar -rcs libA6_30.a A6_30.o
```

After running the mentioned commands, a folder named “**test_outputs**” is created which stores the assembly files and quads output (which stores TAC and Symbol Table). Another folder “**bin”** is created which stores all the assembly executables.

```
To run the executables:

./bin/test<testcase>
```

Design of the Translator

1. **Memory Binding**: This deals with the design of the allocation schema of variables (including parameters and constants) that associates each variable to the respective address expression or register. This needs to handle the following:

   - *Handle local variables, parameters, and return value for a function.* These are automatic and reside in the Activation Record (AR) of the function. Various design schema for AR are possible based on the calling sequence protocol. A sample AR design could be as follows:

Offset	Stack Item	Responsibility
-ve	Saved Registers	Callee Saves & Restores
-ve	Callee Local Data	Callee defines and uses
0	Base Pointer of Caller	Callee Saves & Restores
	Return Address	Saved by call, used by ret
+ve	Return Value	Callee writes, Caller reads
+ve	Parameters	Caller writes, Callee reads

Activation Record Structure with Management Protocol

     - Offset's in the AR are with respect to the Base Pointer of Callee.
     - Return Value can alternatively be returned through a register (like accumulator or `eax`).
     - The AR will be populated from the Symbol Table of the function.

   - *Handle global variables* (note that local static variables are not allowed in nanoC) as static and generate allocations in static area. This will be populated from global symbol table (ST.gbl).
   - *Register Allocations & Assignment*: Create memory binding for variables in registers:
     - After a load / store the variable on the activation record and the register have identical values
     - Registers can be used to store temporary computed values
     - Register allocations are often used to pass `int` or pointer parameters
     - Register allocations are often used to return `int` or pointer values

   Note: *Refer to* Run-Time Environment *lecture presentations for details and examples on memory binding.*

2. **Code Translation**: This deals with the translation of 3–Address quad's to x86 / IA-32 / x86-64 assembly code. This needs to handle:

   - *Generation of Function Prologue*: Few lines of code at the beginning of a function, which prepare the stack and registers for use within the function.
   - *Generate Function Epilogue*: Appears at the end of the function, and restores the stack and registers to the state they were in before the function was called.
   - *Map 3–Address Code to Assembly*: To translate the function body do:
     - Choose optimized assembly instructions for every expression, assignment and control quad.
     - Use algebraic simplification & reduction of strength for choice of assembly instructions from a quad.

   Note: *Refer to* Target Code Generation *lecture presentations for details.*

3. **Target Code**: Integrate all the above code into an Assembly File for gcc assembler.

Shortcomings and bugs

1. Referencing arrays by address has not been handled
2. Function definitions of printStr, printInt and readInt must be present in the code in order to use it