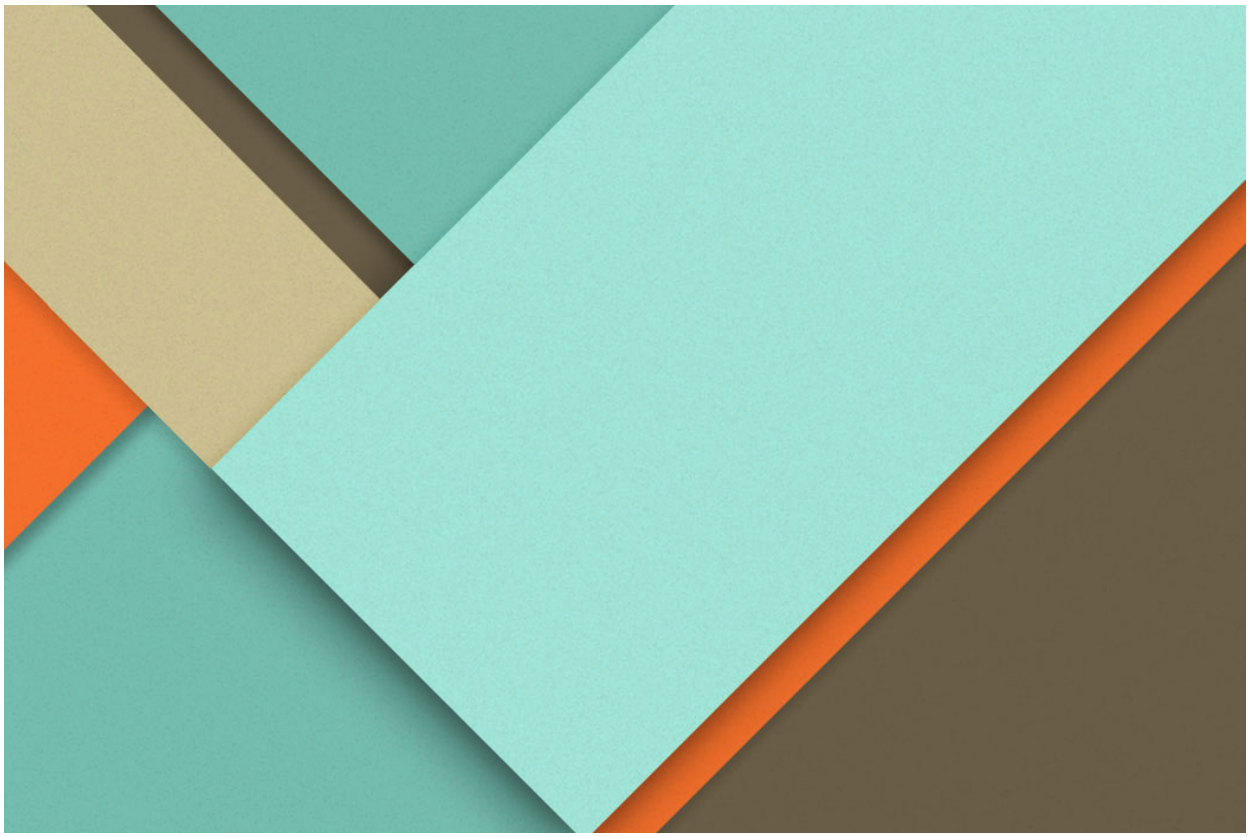

A5- Machine Independent Code Generator for nanoC

24.04.2023

—

Group 30

Aadarshraj Sah (200101001)

Mahek Vora (200101062)

Command to run the program

```
aadarshraj@aadarshraj-HP-ProDesk-600-G5-PCI-MT:~/Documents/Sem6/IPLL/Assgn5/A5_30$ make clean
rm -f lex.yy.c *.tab.c *.tab.h *.output *.o a.out *.gch outputs/*_quads*.out
aadarshraj@aadarshraj-HP-ProDesk-600-G5-PCI-MT:~/Documents/Sem6/IPLL/Assgn5/A5_30$ make test
bison -dtv A5_30.y
flex A5_30.l
g++ -c lex.yy.c
g++ -c A5_30.tab.c
g++ -c A5_30_translator.h
g++ -c A5_30_translator.cpp
g++ lex.yy.o A5_30.tab.o A5_30_translator.o -lfl
mkdir -p ./outputs

Running TAC generator on test input files in ./inputs...

./a.out < inputs/A5_30_test1.c > outputs/A5_30_quads1.out
./a.out < inputs/A5_30_test2.c > outputs/A5_30_quads2.out
./a.out < inputs/A5_30_test3.c > outputs/A5_30_quads3.out
./a.out < inputs/A5_30_test4.c > outputs/A5_30_quads4.out
./a.out < inputs/A5_30_test5.c > outputs/A5_30_quads5.out

Execution successful!

Test outputs generated and written to ./outputs.
```

`make clean`

This removes all the output files:

`rm -f lex.yy.c *.tab.c *.tab.h *.output *.o a.out *.gch outputs/*_quads*.out`

`make test`

It runs the following commands:

`bison -dtv A5_30.y`

`flex A5_30.l`

`g++ -c lex.yy.c`

`g++ -c A5_30.tab.c`

`g++ -c -o A5_30_translator.o combined.cpp`

`g++ lex.yy.o A5_30.tab.o A5_30_translator.o -lfl`

`mkdir -p ./outputs`

After running the mentioned commands, a folder named “outputs” is created which stores the assembly files and quads output (which stores TAC and Symbol Table).

The 3-Address Code

3-Address Code is a low-level intermediate code representation used by compilers to generate machine code. It consists of instructions that take up to three operands and produce a result that can be stored in a variable or a register. The three operands are typically variables, constants or memory addresses.

In the context of the nanoC translator, the 3-Address Code is used as an intermediate representation of the program being translated. It is generated by the translator during the semantic analysis phase and is then used to generate machine code.

Each 3-Address instruction consists of a destination operand, two source operands, and an operator. For example, the statement "x = y + z" would be translated to the following 3-Address instruction:

t1 = y + z

x = t1

Here, t1 is a temporary variable that stores the result of the addition operation.

The 3-Address Code generated by the translator is represented as a sequence of quadruples, where each quadruple consists of four fields: the operator, the first operand, the second operand, and the result. For example, the above 3-Address instruction would be represented as the following quadruple:

+, y, z, t1

=, t1, _ , x

Here, the underscore in the third field indicates that it is not used in this operation.

The 3-Address Code generated by the translator is machine-independent, meaning that it does not depend on the architecture or operating system of the target machine. This makes it easier to generate machine code for different platforms, as the 3-Address Code can be translated to the specific machine code instructions required by the target machine.

Design of the Translator

Lexer & Parser:

- The translator begins with a lexer that reads in the input source code character by character and groups them into tokens. The lexer will recognize keywords, identifiers, literals, operators, and punctuation. The parser then uses the tokens to generate an abstract syntax tree (AST) that represents the structure of the program.

Augmentation:

- The AST is then augmented with additional information, such as the data type of each expression and statement, the scope of each identifier, and any necessary type conversions.

Attributes:

- The AST is further augmented with attributes that represent the 3-address code (TAC) that will be generated for each statement and expression. These attributes will include the operation code, the operands, and any labels or jumps needed for control flow.

Symbol Table:

- A symbol table is created and populated during the parsing phase. It will store information about each identifier used in the program, such as its name, data type, and scope.

QuadArray:

- During the semantic analysis phase, the AST is traversed in depth-first order to generate a quad array. Each quad in the array represents a single TAC instruction, with up to three operands and an operation code.

Global Functions:

- Finally, global functions are generated for the TAC instructions that correspond to functions defined in the input program. These functions will take in the operands for each TAC instruction and perform the necessary operations, such as arithmetic or memory access.

Overall, this design allows for a machine-independent translation of nanoC programs into TAC, with a supporting symbol table and other auxiliary data structures to aid in semantic analysis and code generation.