CPSC-402 Report
Compiler Construction

Aidan Wall
Chapman University

May 19, 2022

Abstract

Short summary of purpose and content.

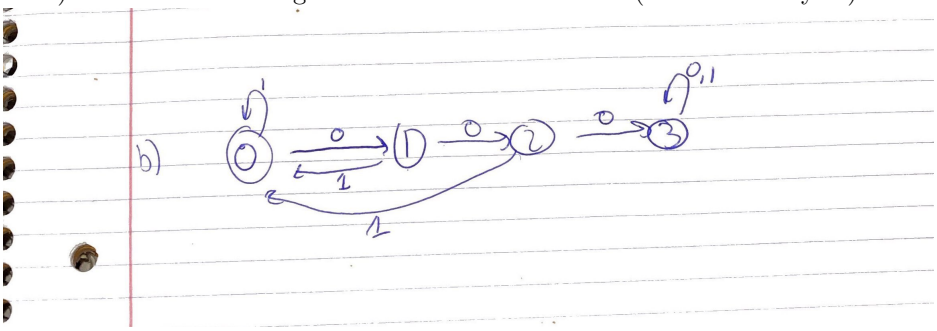
Contents

1 Homework

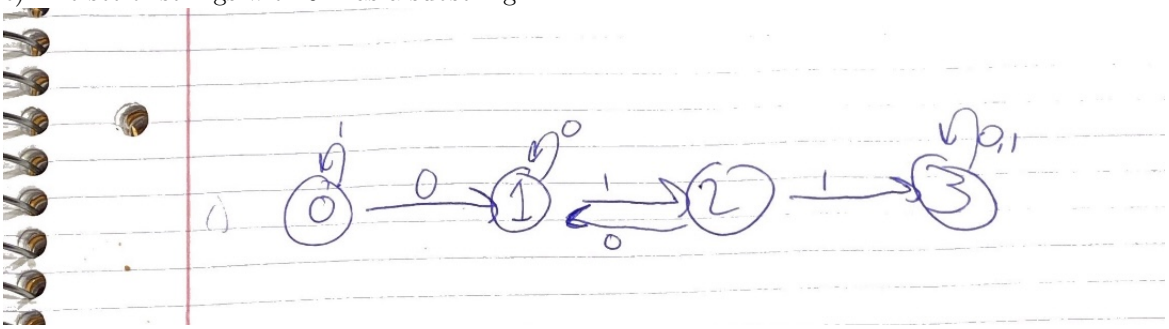
For most weeks, you will have a subsection that contains your answers.

1.1 Week 1

Exercise 2.2.4 Give DFA's accepting the following language over the alphabet 0,1:
b) The set of all strings with three consecutive 0's (not necessarily at)



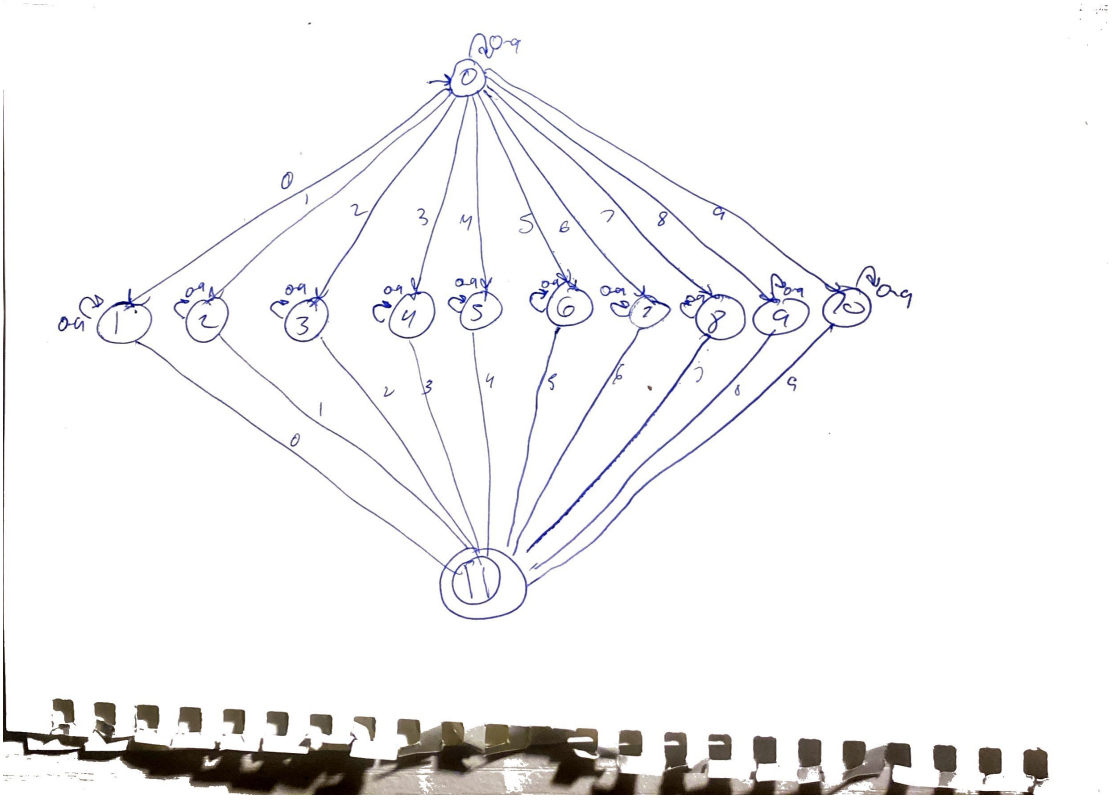
c) The set of strings with 011 as a substring.



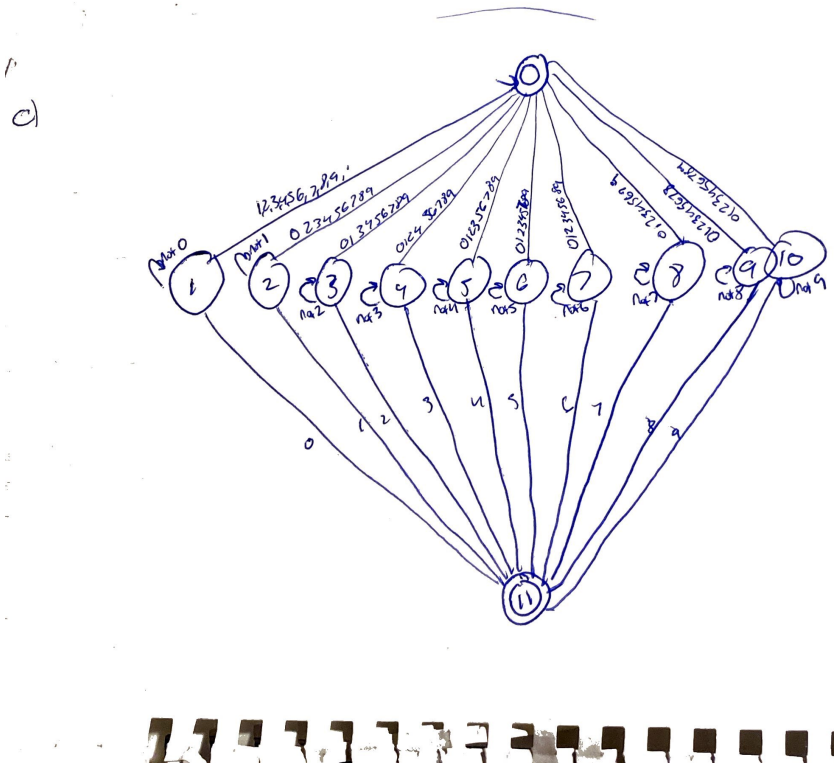
1.2 Week 2

Exercise 2.3.4 Find NFA's that recognize:

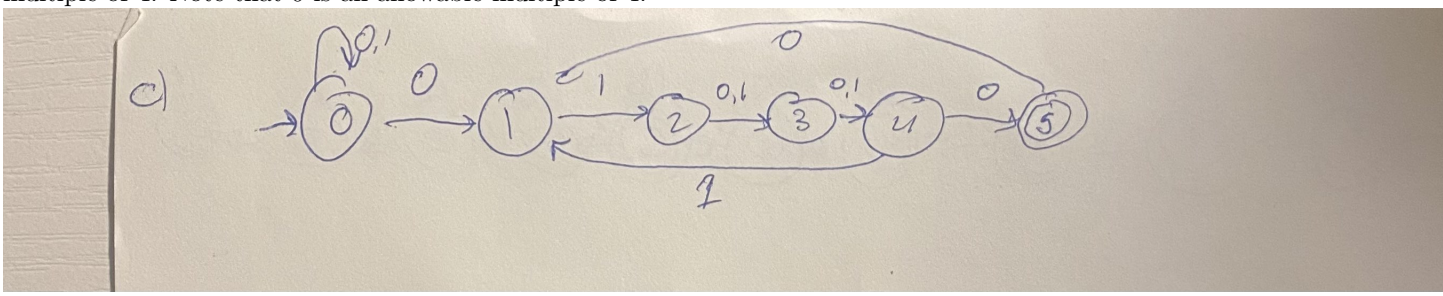
a) The set of strings over alphabet 0,1,...,9 such that the final digit has appeared:



b) The set of strings over alphabet 0,1,...,9 such that the final digit has not appeared before.

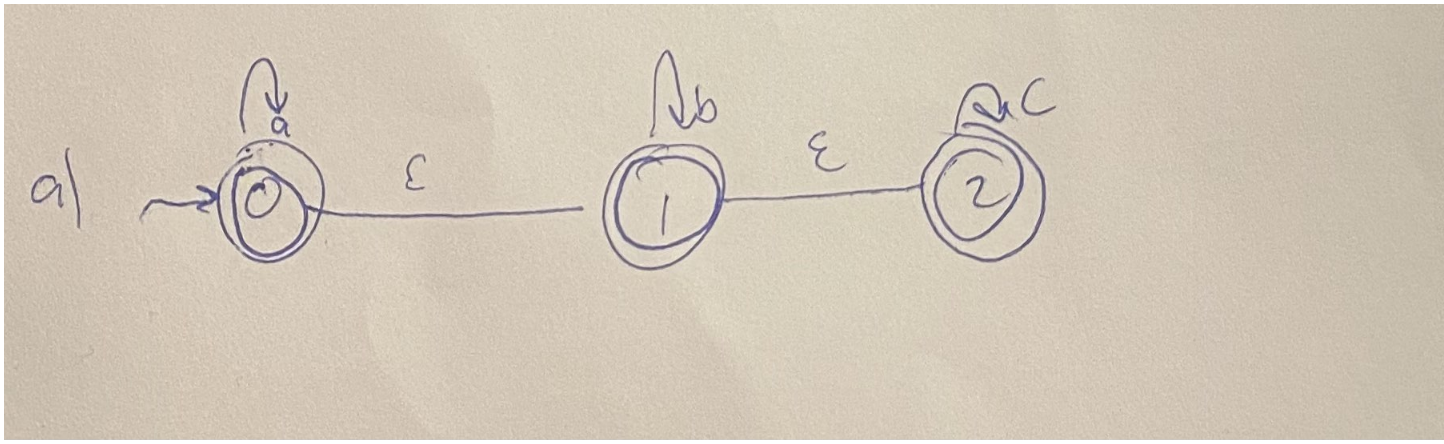


c) The set of strings of 0's and 1's such that there are two 0's separated by a number of positions that is a multiple of 4. Note that 0 is an allowable multiple of 4.

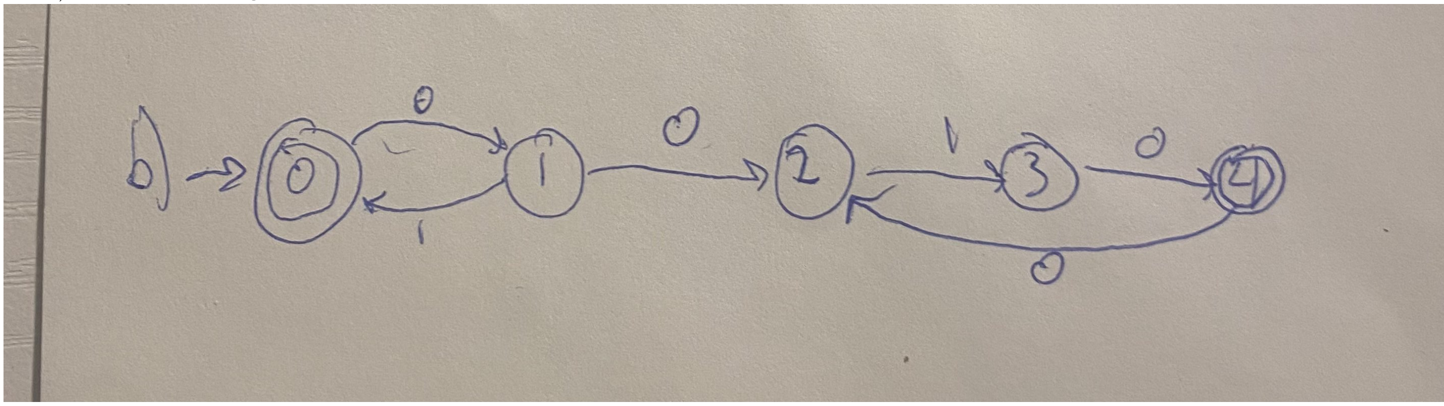


Exercise 2.5.3 Find NFA's that recognize

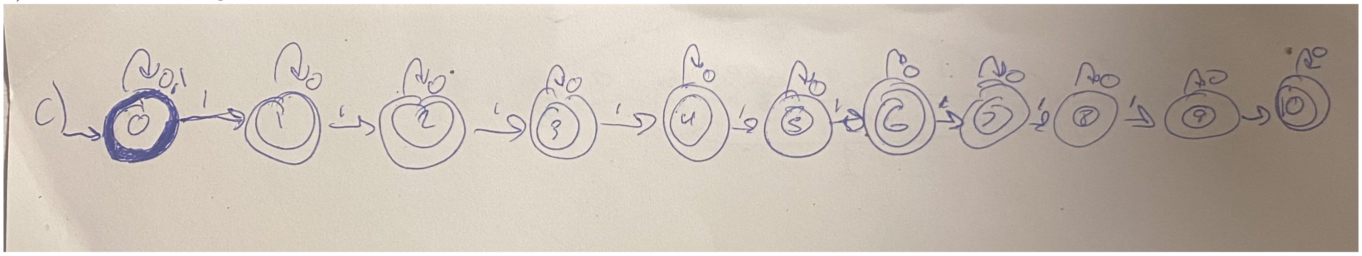
a) The set of strings consisting of zero or more a's followed by zero or more b's, followed by zero or more c's



b) The set of strings that consist of either 01 repeated one or more times or 010 repeated one or more times



c) The set of strings 0's and 1's such that at least one of the last ten positions is a 1

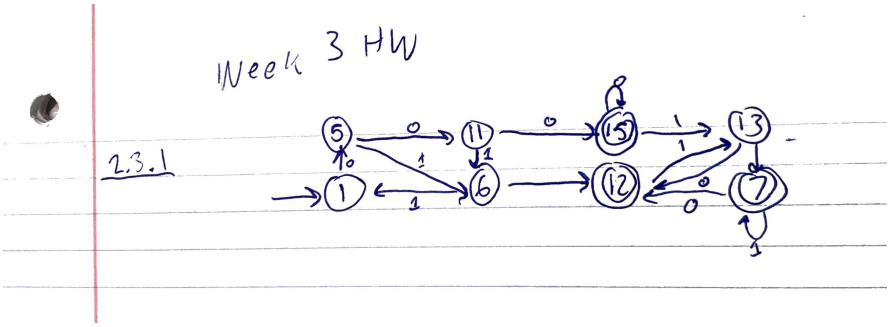


1.3 Week 3

Exercise 2.3.1 Convert to a DFA the following NFA:

	0	1
$\rightarrow p$	$\{p,q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\emptyset$
$*s$	$\{s\}$	$\{s\}$

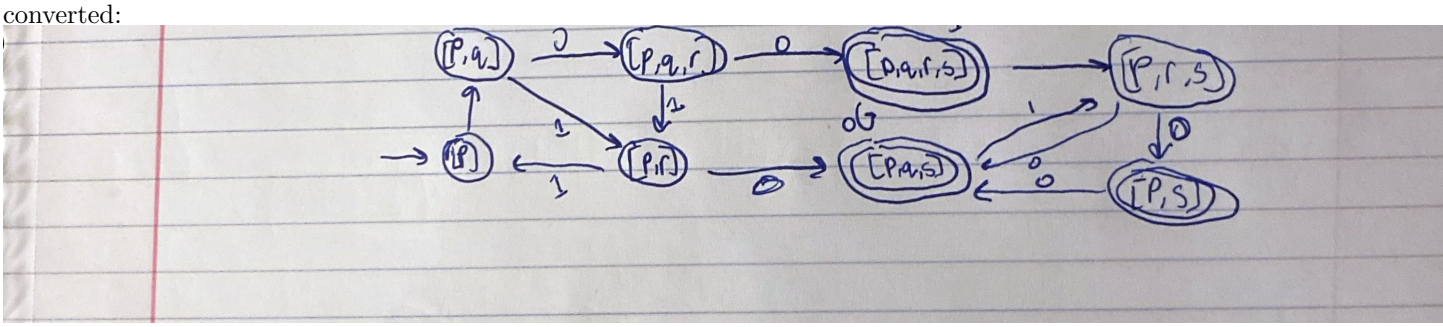
converted:



Exercise 2.3.2 Convert to a DFA the following NFA:

	0	1
$\rightarrow p$	$\{q, s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\emptyset$	$\{p\}$

converted:



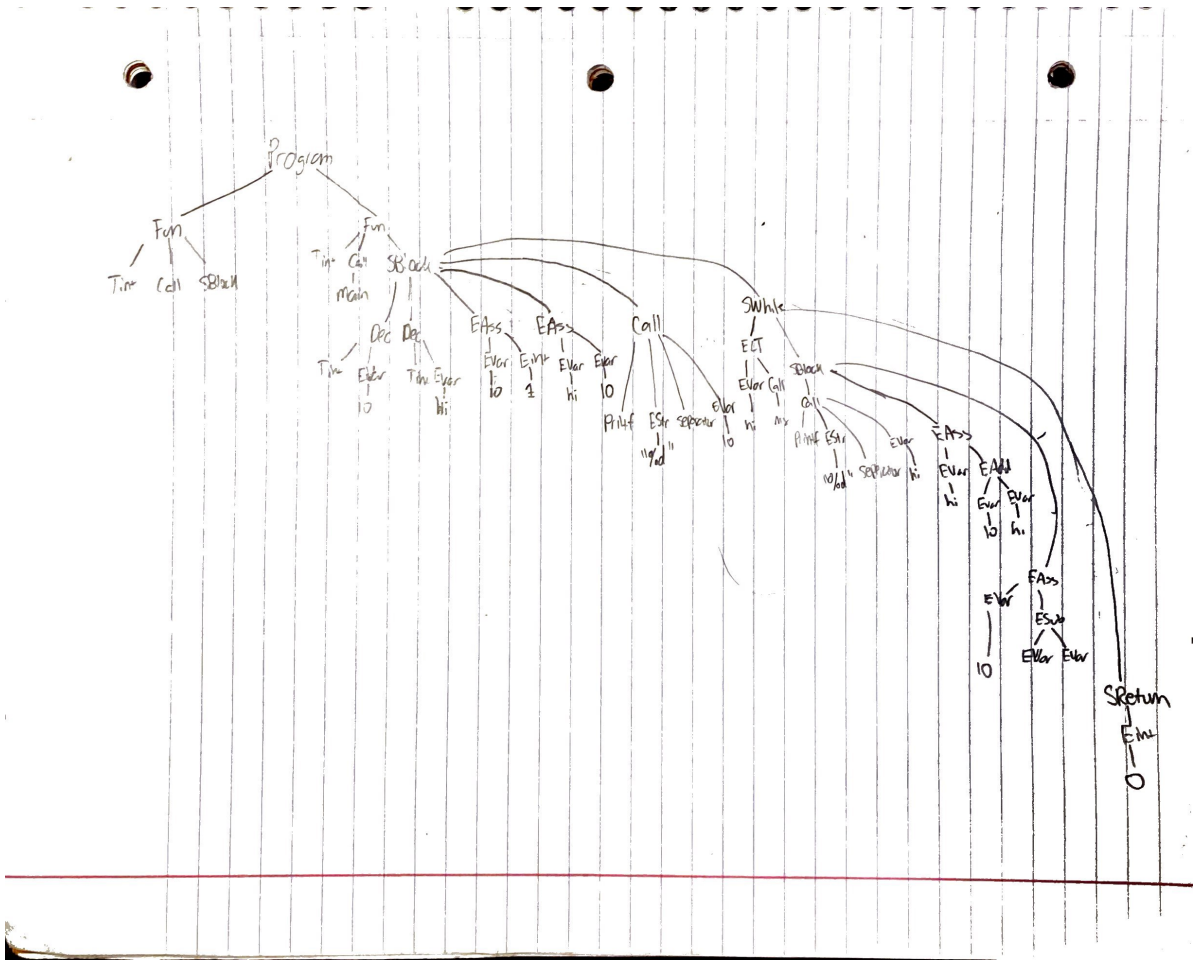
Converting NFAs to DFAs In Haskell Using the List Monad:

```
-- convert an NFA to a DFA
nfa2dfa :: NFA s -> DFA [s]
nfa2dfa nfa = DFA {
  -- exercise: correct the next three definitions
  dfa_initial = [nfa_initial nfa],
dfa_final = let final qs = disjunction(map(nfa_final nfa) qs) in final,
dfa_delta = let
    delta [] c  =[]
    delta (q:qs) c = concat [nfa_delta nfa q c,delta qs c]
    in delta}
```

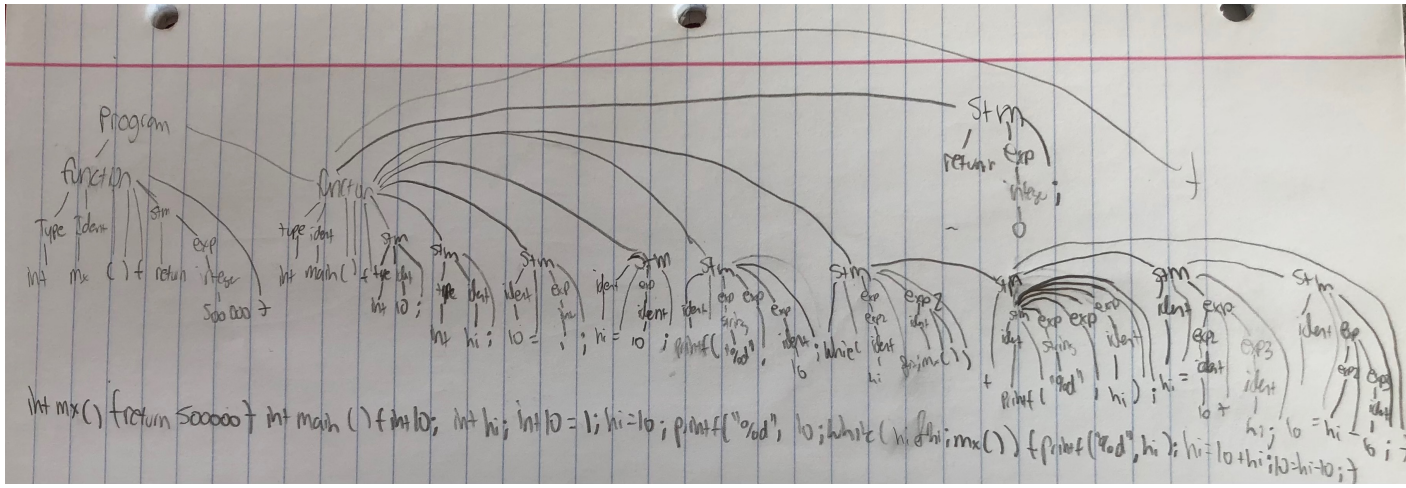
Explanation: The goal is to write an algorithm that will convert an NFA to a DFA. Because the states of the DFA are sets of states of the NFA, we will need to do this many times, which we will do iteratively in dfa_final. dfa_initial has the initial state of the sets, which will contain the initial state of the NFA as list as the only element. dfa_final will have will only be final when it is a final state of the NFA, which we can get from the code already provided. dfa_delta uses the Haskell concat function to concatenate all of the sets together.

1.4 Week 4

Write out the abstract syntax tree for the complete fibonacci program



Write out the parse tree (=concrete syntax tree) for the complete fibonacci program:

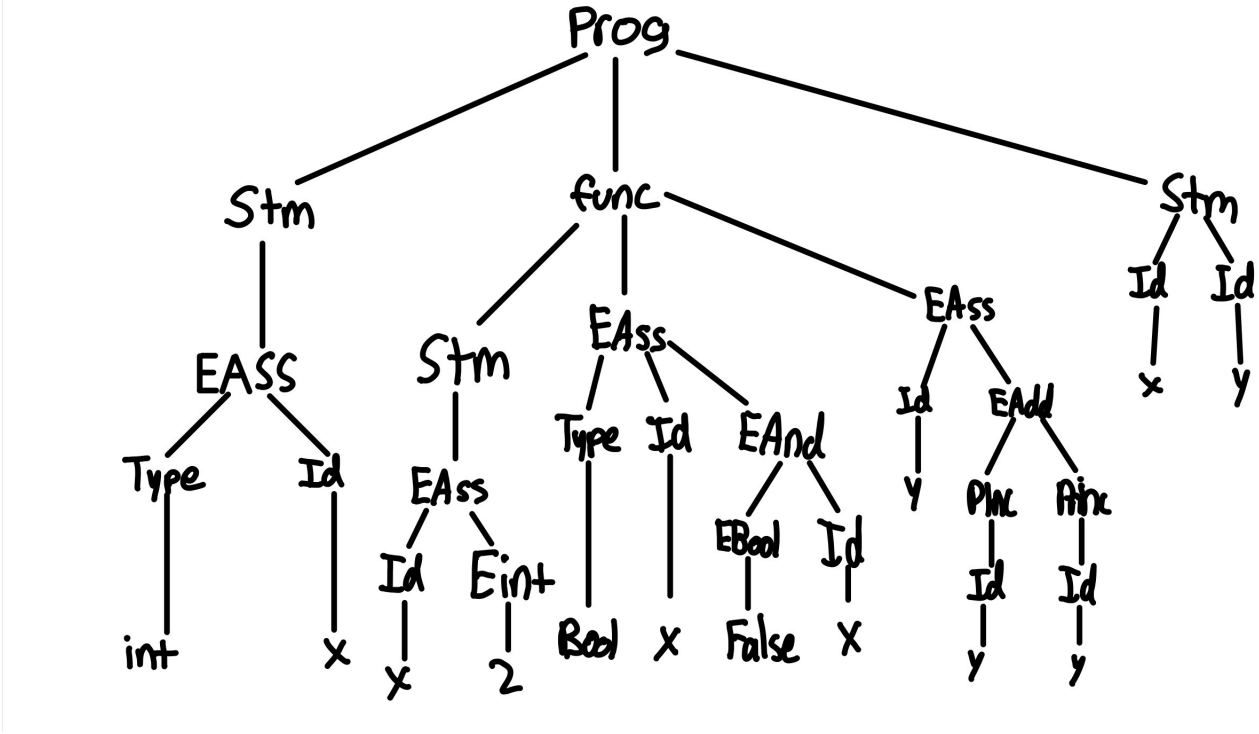


1.5 Week 10

1) Show in the form of a proof tree, the steps taken by an interpreter evaluating the following program fragment:

```
int x;
{
x=2;
bool x = false && x;
y = y++ + ++y ;
}
```

The proof tree for this program fragment:



2 Project

I will be doing the first option of describing and analyzing a compiler in C++ using gcc. I will have a sample program in C++ and convert it to assembly language. I will explain how the compiler is converting the C++ code into assembly language. The program I have chosen will calculate the cube of a number. A cube is a number multiplied by itself 3 times.

Lets first start off by talking about compilers a little bit in general. A compiler is a program that takes programs written in a high level language and transforms them to programs that the computer can understand. It will convert every variable, function, syntax, and everything else. The compiler makes use of a stack to constantly use and update memory registers. A register is simply a small bit of memory inside of the CPU. They are used by assembly language and can perform a variety of tasks. Registers accept, store, and transfer data and instructions from the CPU. All of these registers and data are stored on the stack. The stack is really useful when controlling registers and subroutines because it is very easy to keep track of the point where each active process should return control when it has completed its task.

The program that I am going to convert into assembly is as follows:

```
#include <iostream>
using namespace std;

int cube(int num){

return num * num * num;

}

int main () {
return cube(3);
}
```

The most basic version of our program, is just an empty function that returns 0. We I have modified the code so that my cube function just returns 0. It looks like this:

```
int cube(){
return 0;
}
```

This is what the output looks like in assembly:

```
cube():
    push   rbp
    mov    rbp, rsp
    mov    eax, 0
    pop    rbp
    ret
```

[GODBOLT] When looking at the first line of both the C++ code and the assembly code we see that the first thing in both is the function cube()[FASS]. This is called the header in assembly.

The next line: **push rbp**. rbp registers are special purpose registers, which is the base pointer, which points to the base of the current stack frame. By pushing the rbp register first, it puts this register at the top of the stack.

The next line **mov rbp, rsp**, now has a rsp register, which is the stack pointer, which points to the top of the current stack frame[ASRC]. This line copies the value of the stack pointer rsp to the base pointer rbp, so that now both of these registers point to the value on the top of the stack [SARC].

The next line **mov eax, 0** uses a general purpose register eax. This line copies the value 0 into the eax register. In this version of assembly, a functions return value is stored in the eax register. This function sets the return value to 0 at the end of our function. This is directly the line **return 0;** in our C++ code [ASRC].

The second to last line **pop rbp** pops off the base pointer that is currently on the top of the stack, off the stack, and stores it back into the rbp register. The rbp has to be popped off because we pushed it earlier in the program. Anything that is put on the stack also has to be removed from the stack, otherwise the program will crash.

The last line **ret** pops the return address from the top of the stack. This happens when functions are

called, and the program uses this call to push the return address before it actually executes any of the instructions in the function.

Now that we have seen the bone structure for a simple conversion from a C++ function to assembly, next I will convert the entire function to cube a number into assembly, and can take a look at the difference:

The code for our C++ function is as follows:

```
int cube(int num){
return num * num * num;
}
```

The outputted assembly code for this C++ function is as follows:

```
cube(int):
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp-4], edi
    mov    eax, DWORD PTR [rbp-4]
    imul   eax, eax
    imul   eax, DWORD PTR [rbp-4]
    pop    rbp
    ret
```

[GODBOLT]
There are a couple differences in output between the basic empty function and this function. In the header, **cube(int)** there is now an being passed in to the function. It is of the type int and is the variable **num** in our C++ function. The next two lines **push rbp** and **mov rbp, rsp** are the same as the previous program. They just set the register to the top of the stack and points the registers to it.

The next line **mov DWORD PTR [rbp-4], edi** is one that was not in the previous program but is now. This line is in reference to the argument passed to our cube function. It is storing the integer argument num, into the edi register. An edi register is a destination index register used for strings, memory array copying and setting. This line is copying the num argument locally (with an offset of -4 bytes from the frame pointer stored in the rbp [PTR].

The following line **mov eax, DWORD PTR [rbp-4]** copies the value in the local register to the eax register.

The next line **imul eax eax** uses the built in imul function, which executes a signed multiplication of a register by a register and stores the product in a register [IMUL]. In this case, it is multiplying the eax register, which has the stored value of the passed "num" argument, with itself, the eax register.

The previous line squared the number, but the number needs to be cubed. The line **imul eax, DWORD PTR [rbp-4]** multiplies the already squared value of eax, with the value at the position of rbp-4, where the original value of our argument was stored[IMUL]. This completes the cubing of the number, and **pop rbp** is called to store the value back in the rbp register. This is the same as the previous example, and will always be done when it is pushed earlier in the program. The last line pops the return address from the top of the stack and completes the program.

The next part, and the most important, of converting a C++ code to assembly is the main function. To start from the beginning, I will explain a simple main function so we can understand how the framework works. Our C++ main function does nothing and returns 0:

```
int main() {
return 0;
  }
```

This simple main function in C++ in assembly is:

```
main:
    push   rbp
    mov    rbp, rsp
    mov    eax, 0
    pop    rbp
    ret
```

[GODBOLT]

The only difference between this function and the one I showed earlier is the header is different. This one is header for the program called main instead of cube.
When combining all of this together, we declare the cube function, then build our main and call the cube function on the number 5. The final C++ program looks like this:

```
int cube(int num) {
    return num * num * num;
}

int main(){
    return cube(5);
}
```

The final output for our assembly code is as follows:

```
cube(int):
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp-4], edi
    mov    eax, DWORD PTR [rbp-4]
    imul   eax, eax
    imul   eax, DWORD PTR [rbp-4]
    pop    rbp
    ret
main:
    push   rbp
    mov    rbp, rsp
    mov    edi, 5
    call   cube(int)
    nop
    pop    rbp
    ret
```

When looking at this output of main vs the previous output of main, there are 3 differences with 2 of them being additional lines. In the previous main, the value 0 was set to the eax register. In this, the int value 5 is being set to the edi register. The eax register used before is for temporary data storage and memory access. The edi, destination index register, is being used here instead, with the value 5 being given to it [x86]. There are also two additional lines that were not there before, they are:

call cube(int)

nop

The line **call cube(int)** uses the keyword "call" and calls the function cube, with an int as an argument. The function can later return using 'ret' at the end. Call will store the return address to jump back onto the stack [FASS].

The next new line **nop** is a no operation, which is an instruction that doesn't implement any sort of operation.

no-op instructions are used for timing purposes. They can also help to deal with certain memory issues, or work in conjunction with a group of other instructions to facilitate some kind of end result. No-op instructions may be used for pipeline synchronization or to delay some type of CPU activity.

A no operation or "no-op" instruction in an assembly language is an instruction that does not implement any operation [TECHNOPEDIA]. No-op's are often used for timing purposes, often to delay an instruction, or to help with memory issues, and a variety of other things.

The code ends as how every other one of the assembly code has ended. Popping the rpb register with the stored value of 125 (5*5*5) and then returns it, breaking the call function and terminating our program.

3 Conclusions

Compilers are the magical bridge between the brilliance of programmers and the brilliance of computers. They allow programmers all over the world to write code in high level languages and build things with ease. If programmers had to interact with machine code themselves and did not have compilers, we would not be where we are today technologically. They bridge the gap between software and hardware, and allow people to specialize in one aspect of this so that they can be more efficient. However, building a compiler is no easy task. It draws ideas from discrete math, linear algebra, programming language theory, computer architecture, electrical engineering, and so many other disciples. Compilers have been a very important important of scientific history, and have allowed us to do so much. I am glad that I got to learn more about the intricacies of a compiler, and it has made me view programming in a different light. I now understand different errors better, and what I can and cannot do when writing code.

References

[HMU] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman: Introduction to automata theory, languages, and computation, 3rd Edition. Pearson international edition, Addison-Wesley 2007

[CPP] Intro C++ Program

[FASS] Calling Functions in Assembly

[IMUL] Oracle IMUL Documentation

[x86] x86 Registers

[PTR] Understanding Assembly in C

[GODBOLT] C++ to web assembly

[SARC] Stack Register Assembly Code

[ASRC] Understanding C by Learning Assembly

[TECHNOPEDIA] What Are "No-Ops"?