CPSC-402 Report
Compiler Construction

Samuel Ellenhorn
Chapman University

May 22, 2022

Abstract

The purpose of this paper is to demonstrate my knowledge the compiler which sits behind high level code. Proper knowledge of assembly code can, in many cases lead to extreme improvements in efficiency. It is also possible that some tasks can only be completed when implemented in assembly code.

Contents

1 Introduction

1.1 General Remarks

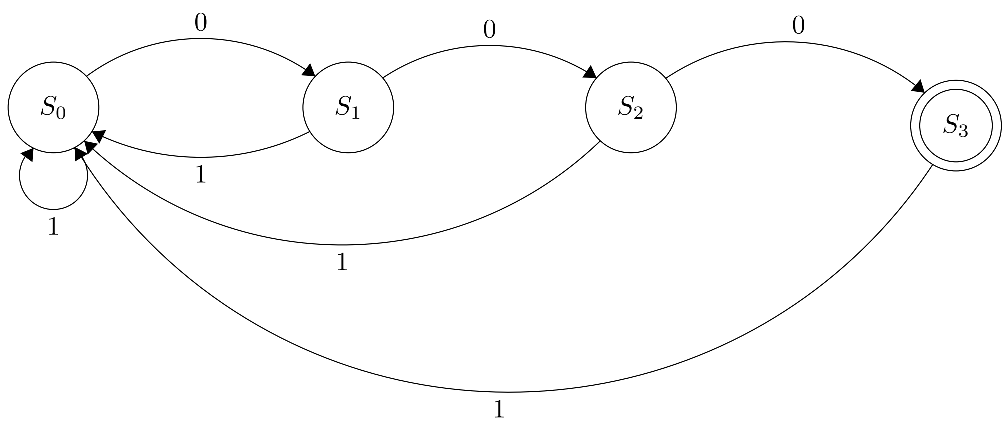
Compiler construction is an interesting course which I have enjoyed. I have found some of the assignments to be very difficult, however, after completion I have gained insight into creating more efficient code. The following report contains examples of my work throughout the year.

2 Homework

2.1 Homework 1

Exercise 2.2.4 Give DFA's accepting the following languages over the alpahbet 0, 1:
a) The set of all strings ending in 000

Answer:



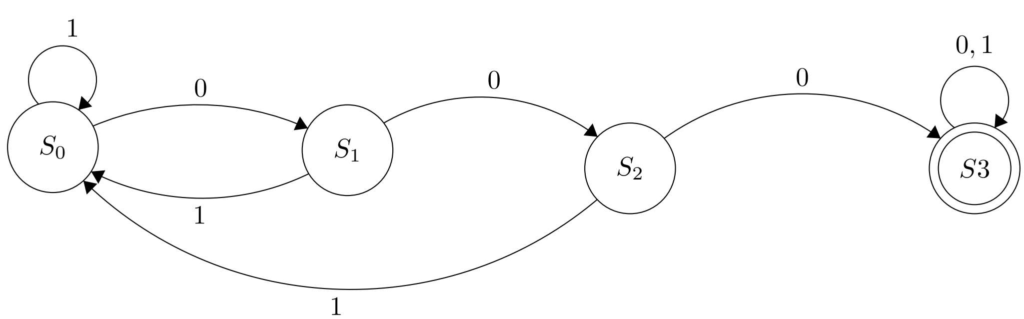
b)The set of all strings with three consecutive 0's(not necessarily at the end):

Answer:

This language is described as follows:

$$L = \{w\epsilon\{0,1\}^*|w = x000y, x, y\epsilon\{0,1\}^*\}$$

A DFA for this language can be seen here:



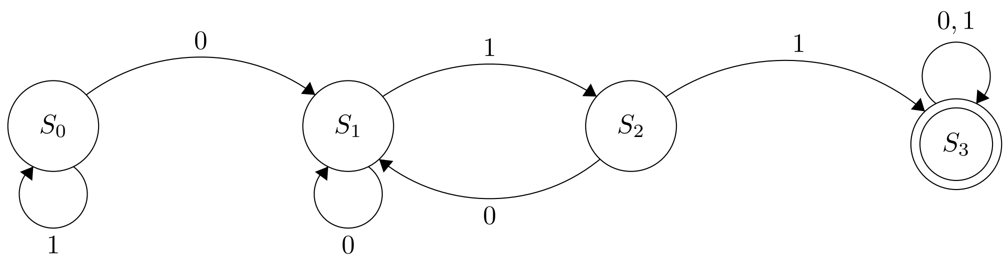
c)The set of strings with 011 as a substring.

Answer:

This language is described as follows:

$$L = \{w\epsilon\{0,1\}^*|w = x0y1z1u, x, yz, u\epsilon\{0,1\}^*\}$$

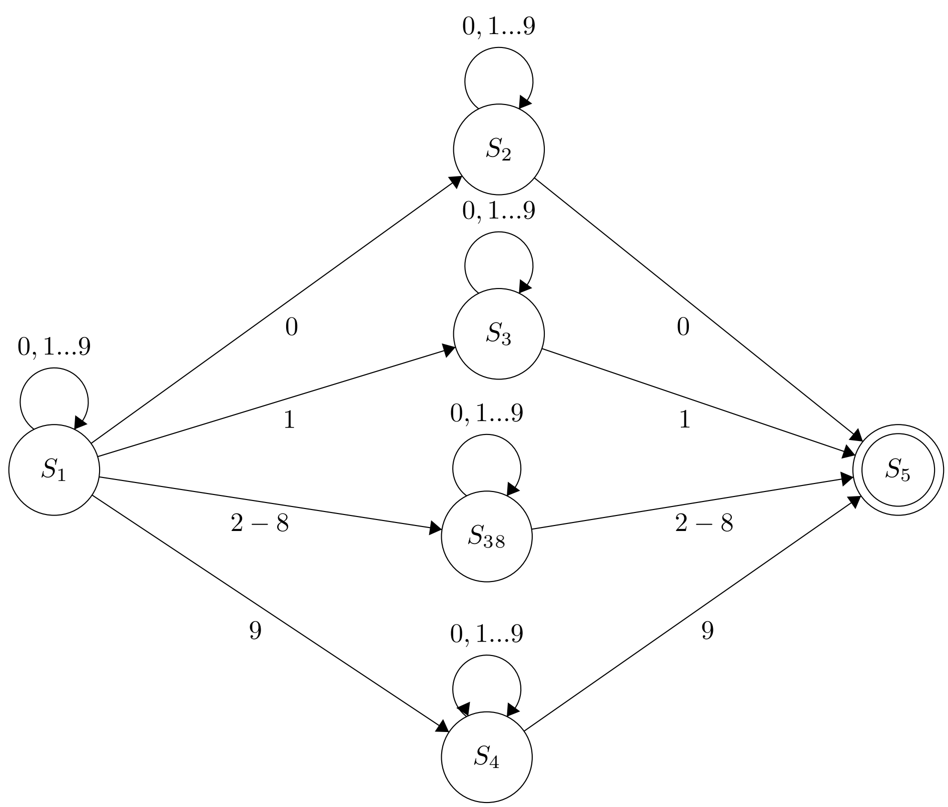
A DFA for this language can be seen here:



2.2 Homework 2 with Regular Expressions

Exercise 2.3.4
a) The set of strings over alphabet 0,1...9 such that the final digit has appeared before.

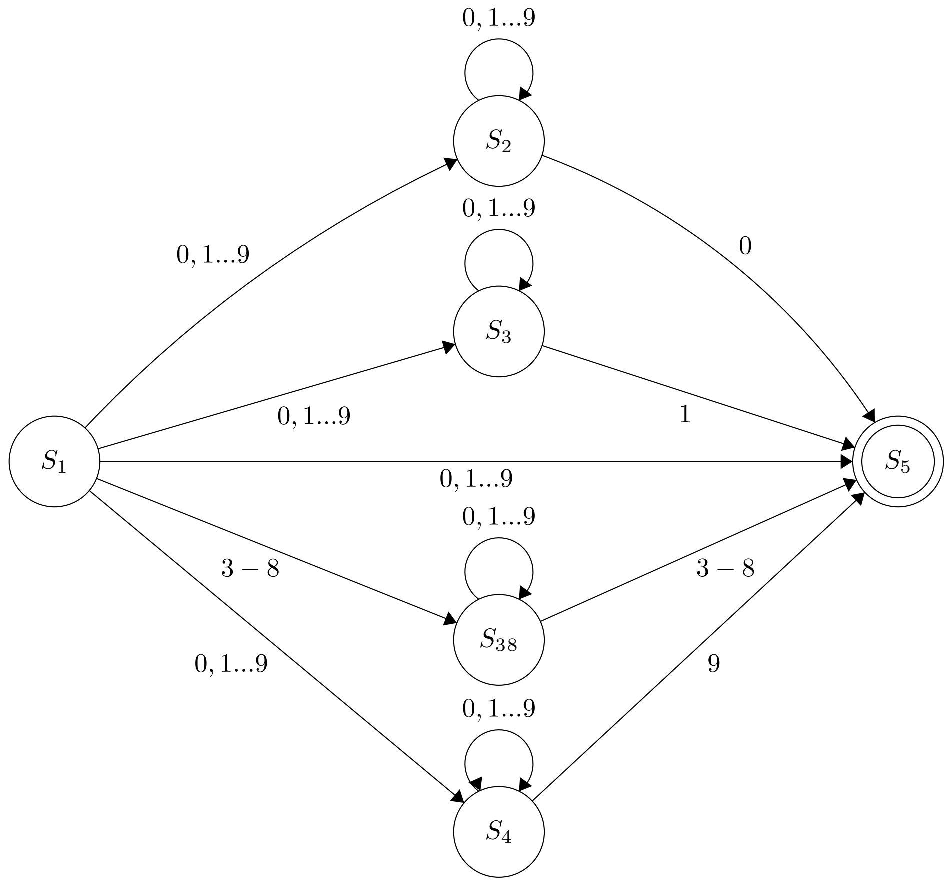
Answer:



Regular Expressions:
$\{0 + ...9\}^*(0\{0 + ...9\}^*0 + ...9)$

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

Answer:

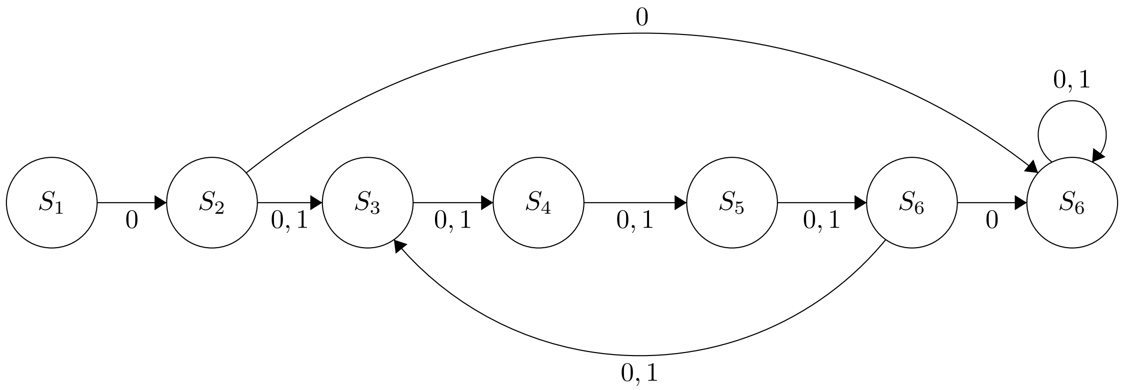


Definitions:
$\sigma_0 = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$
$\sigma_1 = 0 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$

First answer:$\{0 + ...9\}^*(0\{0 + ...9\}0 + ...)$
Final Answer: $\sigma^*0 + \sigma^*1 + \sigma^*2 + \sigma^*3...\sigma^*9$

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

Answer:

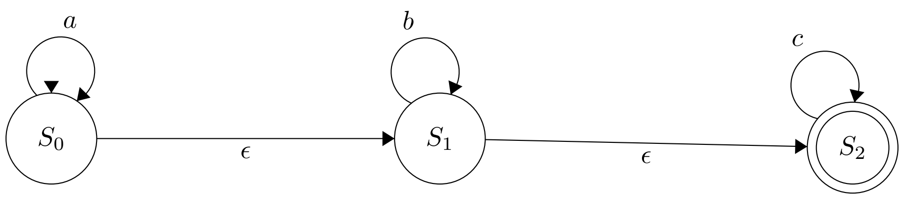


Regular Expressions: $(0+1)^{*}0((0+1)(0+1)(0+1)(0+1))^{*}(0+1)^{*}$

Exercise 2.5.3

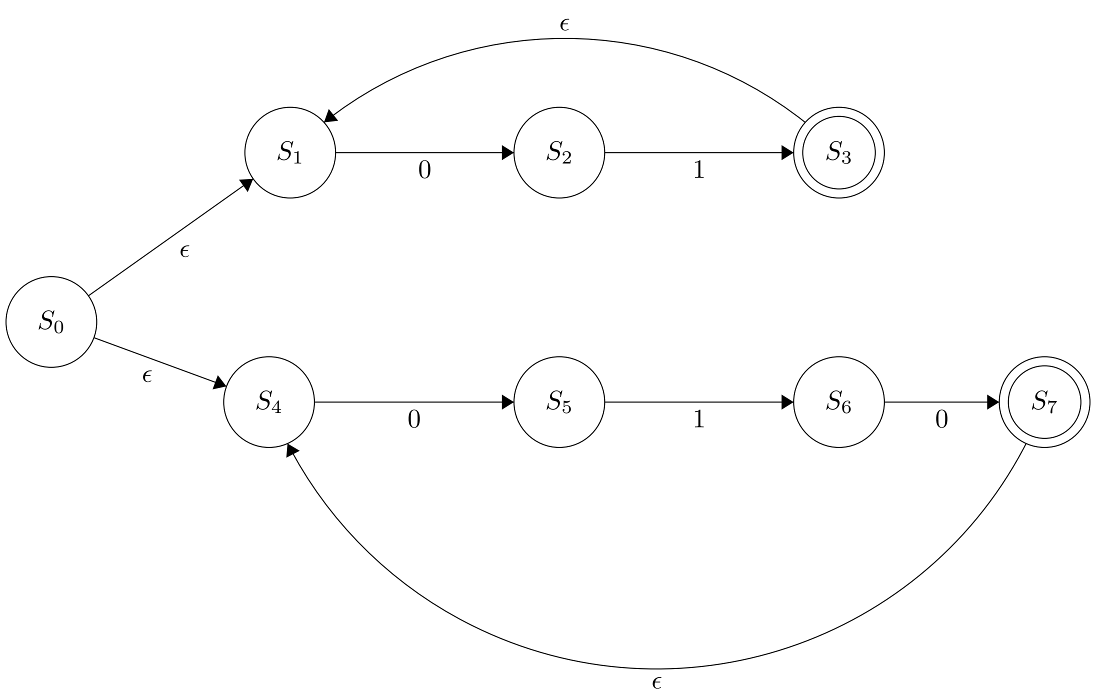
a) The set of strings consisting of zero or more a's followed by zero or more b's followed by 0 or more c's

Answer:

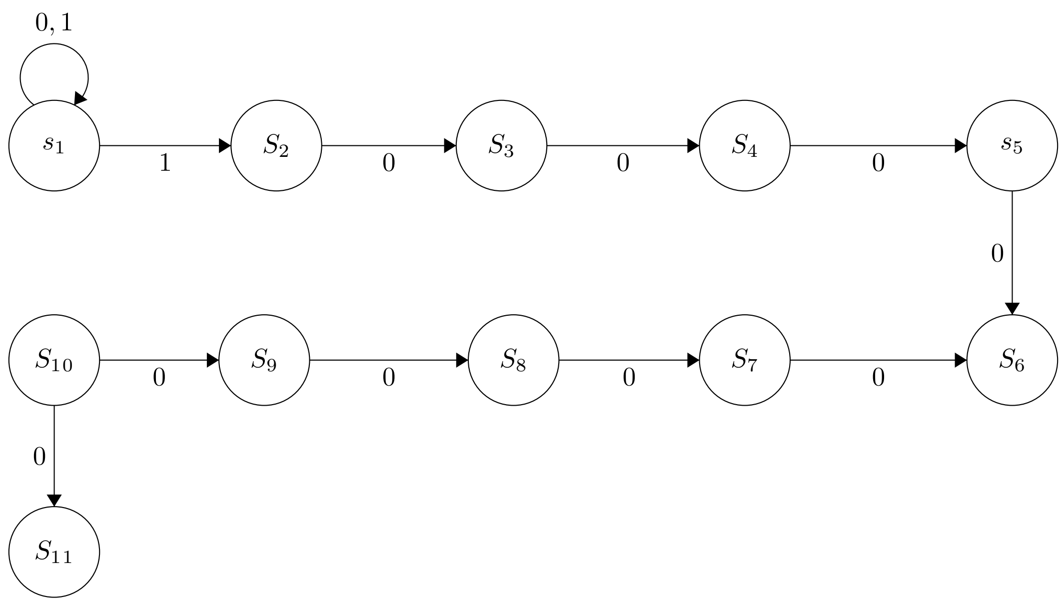


Regular Expressions: a* b* c*

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



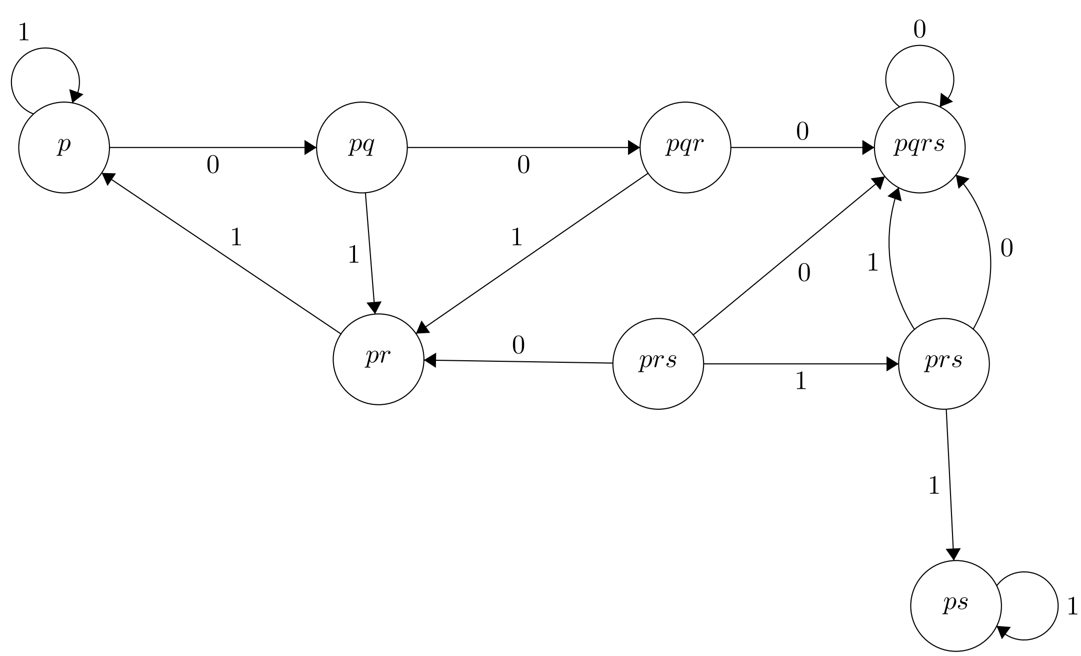
Regular Expressions: $01(01)^* + 010(010)*$
c) The set of strings of 0's and 1's such that at least one of the last ten positions is a 1.



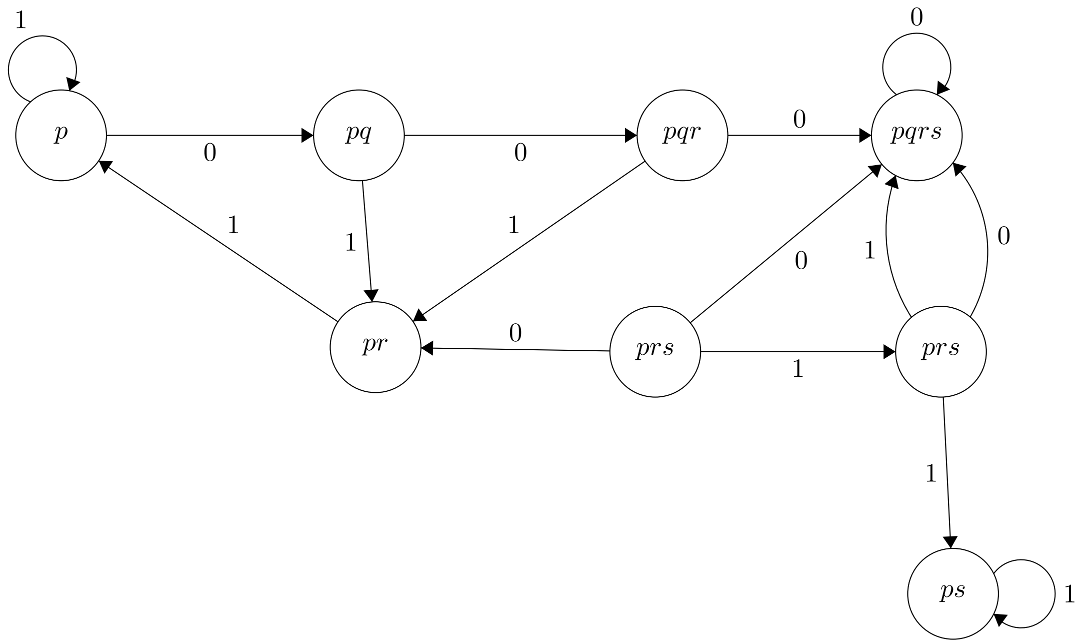
Helper definition: $0^n = 000...0$ (n zeros)
Regular Expression:
$1 + 10 + 100 + 10^3 + 10^4 + 10^5 + 10^6 + 10^7 + 10^8 + 10^9$

2.3 Homework 3: Converting NFA's to DFA's

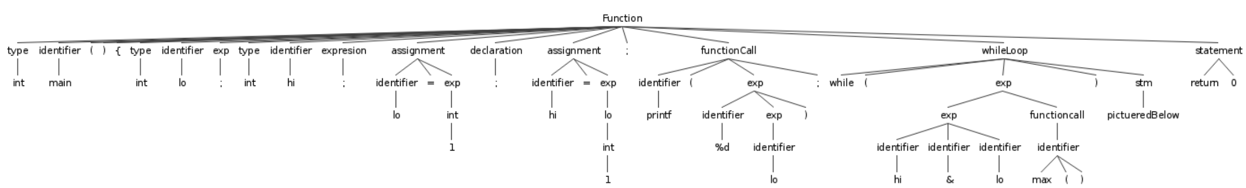
2.3.1

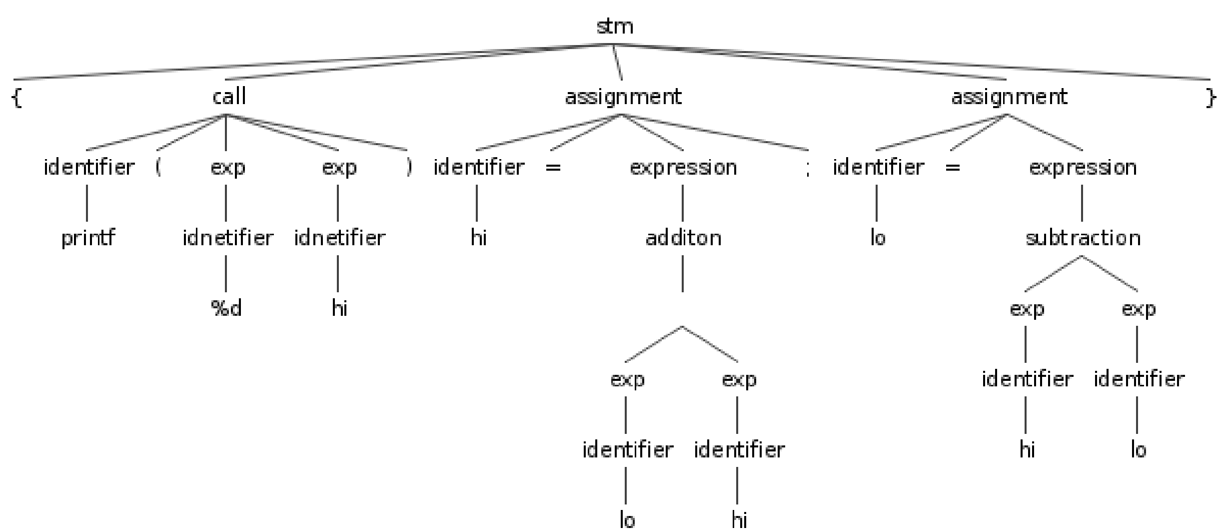


2.3.2



2.4 Homework 4: Parsing a Program by Hand





2.5 Homework 5: Proof Tree

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

Will produce this environment :

G= [x := T]
G_n = [x:=2]
G_1 = [x:=bool]
G_2=[]---y is never declared?
G_2=[v:= y+1]
G_2=[b:= 1+y]
G_2=[y:=v+b]

```
                                                                                 G_2
                                                                               -------   ----
                                                G_1.[] ===>                      y++      ++y
                                              -----------------------------------------------------------------
G.[]===> 2    DA      <2, G.[]>               G_1.[]  ===>  false && x;          ===>y++ + ++ y
-------------------------------------------------------------------------------- ----------------------------------
G.[]===>x=2;    DA    <2, [x:=2].[]>          G_n.[]  ===>bool x = false && x;   ===> y = y++ + ++y;  G_1=G_2
--------------------------------------------------------------------------------------------------------------------
G.[] ===>x=2; bool x = false && x; y = y++ + ++y;}                               ===> y= y++ + ++y   == >x=y;
--------------------------------------------------------------------------------------------------------------------
[x:= T] ===> {   x = 2 ;  bool x = false && x ;  y = y++ + ++y ; }  x = y ;
--------------------------------------------------------------------------------------------------------------------
===>  int x ; {   x = 2 ;  bool x = false && x ;  y = y++ + ++y ; }  x = y ;
```

3 Project

For my project I am choosing to compile C++ into assembly code using the gcc compiler. In order to accurately demonstrate some of the functionality of gcc, it is important to touch on some concepts. The following are components of compilation which I will speak about: function call, variable declaration, conditionals, and some basic computations. I will be working my way towards a program in c++ which will calculate whether or not a number is a palindrome.

In order to get started it is important to figure out what is most basic and essential to a programming language. Most programming languages allow for the declaration of scoped or global variables. The palindrome program is no exception. What can be seen in the code below is four declarations of integers located inside a main.

```
int main()
{
    int n, num, digit, rev = 0;
}
return 0
```

The corresponding assembly code can be seen here:

```
main:
       push   rbp
       mov    rbp, rsp
       mov    DWORD PTR [rbp-4], 0
       mov    eax, 0
       pop    rbp
       ret
```

The functions push and mov are called at the beginning of a new context or scope. The push function pushes to the stack in memory and is called on the base pointer of the context in this case. The mov instruction copies data from the second argument into the first argument, which in this case is now the stack pointer.

In contrast, the functions pop and ret are called at the end of every context. In this case the pop function pops the first item from the stack while the ret function clears the information regarding the stack located in system hardware. From this, you can conclude that there are two arguments which correspond to assigning the four variables.

It is important to note that the push function decrements the stack pointer register by 4, then places its second argument into the 32 bit location of the first argument. As a result, the function call mov DWORD PTR [rbp-4], 0 will yield the the 32-bit integer representation of 0 being moved into the 4 bytes starting at the address in rbp-4. The line: mov eax, [0] results in the movement of the 4 bytes in memory at the address contained in 0 into EAX.

3.1 Do While Loop

In this variation of the code, I will explore how the gcc compiler deals with Do while loops. In addition the code below exemplifies the difference between inline declaration and declarations on multiple lines. For expressions of the same type that are declared on there own line, Assembly code will individually call the mov DWORD PTR call to a separate location. This is why it is more efficient in most cases to use inline declaration. The following code is an example of a do while loop in c++:

```
int main()
{
    int n, num, digit, rev = 1;
    int x = 5;
   do
    {
        num = num + 3;
    } while (num != 1000);
    return 0;
}
```

The corresponding assembly code can be seen below:

```
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-8], 1
        mov     DWORD PTR [rbp-12], 5
.L2:
        add     DWORD PTR [rbp-4], 3
        cmp     DWORD PTR [rbp-4], 1000
        jne     .L2
        mov     eax, 0
        pop     rbp
        ret
```

From the code above you can see that a new context is created for the do while loop. In line two, you can see that DWORD PTR [rbp-4], which refers to the value num is incremented by 3 using the add command. An interesting discovery I made is that you can find information about the exact location of each variable by compiling with different values in place of num. This provides in site into how the DWORD PTR command works.

After the add command is ran, the cmp command is used to compare equality between num and the value 1000. If the result is not equal, the subsequent line jne will trigger a jump back to the beginning of the do loop. If the values are equal, it will then exit the loop. The last three command mov, pop rbp, and ret are used to clear the context and maintain scope.

3.2 if else

The if else statement is central to almost any coding language. Because of this, it makes sense to try to understand what is occurring at the assembly level code. The following is a simple example of an if else statement as well as the corresponding assembly code.

```
int main()
{
    int n, num, digit, rev = 0;

    if (n == rev)
       num++;
    else
        digit++;
   return 0;
}
```

The corresponding assembly code:

```
 main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 0
        mov     eax, DWORD PTR [rbp-8] \\n is at location rbp-12
        cmp     eax, DWORD PTR [rbp-4] \\eax corresponds to n
        jne     .L2
        add     DWORD PTR [rbp-16], 1 \\num is at location rbp16
        jmp     .L3
.L2:
        add     DWORD PTR [rbp-12], 1 \\digit is at location rbp-12
.L3:
```

```
        mov     eax, 0
        pop     rbp
        ret
```

In this assembly code, there are three contexts to examine. First, the context labled .L3 deals with the return statement. .L3 is only called after the else clause. Context .L2 corresponds to the line where digit is post incremented. The if clause is evaluated for equality on lines 5 and 6 with the cmp and jne command respectively. If the jump is not executed, the add statement is executed. The add statement corresponds to the evaluation of the else clause. Utilizing this method, the gcc compiler is able to generate assembly code which function's properly. Because compilation works by recursion over abstract syntax, the compiler must be clever when computing math with re-declared variables. This can be seen in the following section.

3.3 Basic Computations

In C++ there are only 5 lines of simple code to understand. In assembly the code is considerably more complicated. Because of this it is useful to break the code into each computation. The following section will refer to this block of code broken down:

```
int main()
{
    int n, num, digit, rev = 0;
    digit = num % 10;
    rev = (rev * 10) + digit;
    num = num / 10;
    return 0;
}
```

3.3.1 Modulo

The following code is what is generated as a result of the second line of code in the section above. This single line of code in C++ results in a complicated set of movements for assembly code. The following code exemplifies how assembly deals with the modulo operation successfully.

```
        mov     edx, DWORD PTR [rbp-8]
        movsx   rax, edx            \\rax contains num
        imul    rax, rax, 1717986919  \\rax=rax*1717986919=8589934595 (all ones in binary)
        shr     rax, 32       \\unsigned divide
        mov     ecx, eax
        sar     ecx, 2        \\signed divide
        mov     eax, edx  \\edx is still num which is 0
        sar     eax, 31    \\signed divide
        sub     ecx, eax
        mov     eax, ecx
        sal     eax, 2 \\multiplies
        add     eax, ecx
        add     eax, eax
        sub     edx, eax
        mov     DWORD PTR [rbp-12], edx
```

The algorithm used for modulo is a bit complicated. First, the value num is placed into the register rax. Then, imul is called with three arguments resulting in the multiplication of the second and third arguments. The result of imul is then stored in the first argument. The shr will shift the rax register by 32 bits. The operation sar then acts as division in this case. The SAR command is then called with a value of 31. The value of the following is subtracted from ecx and stored in eax. the sal command is the called on eax with a value of 2. The values of ecx and eax are added and stored in eax. The desired result is then acquired by subtracting eax from edx and storing the final result in edx.

The key to understanding how this functions is by understanding the relationship between sar, sal and shr. The command sal shifts a value left based on the second input. A left shift in binary is similar to multiplication by power of 2. In contrast the sar and shr commands act as a right shifts. The shift proportionally to the second operand. A left shift will result in division by a power of 2.

By utilizing this algorithm, the assembly code can successfully calculate modulo and division.

3.3.2 Multiplication

The following line requires a variable be re declared, as well as two variable being added. While this may be easy in C++, it takes considerably more effort in assembly.

```
        rev = (rev * 10) + digit;
```

The corresponding assembly code can be seen here:

```
mov     edx, DWORD PTR [rbp-4]
        mov     eax, edx
        sal     eax, 2
        add     eax, edx
        add     eax, eax
        mov     edx, eax
        mov     eax, DWORD PTR [rbp-12]
        add     eax, edx
        mov     DWORD PTR [rbp-4], eax
```

The value of rev is obtained from DWORD PTR [rbp-4] with the mov function call. The next three lines are the main algorithm which accomplishes our desired computation. The sal instruction shift the bits in the eax register left. The second argument of sal refers to the number of bits to shift. The next two lines shift the bits by a value of two in eax, aswell as add eax to our original value and itself. This results in successful computation of rev * 10. The last three lines move registers and add digit to the previously calculated informant. the result is stored at DWORD PTR [rbp-4].

3.3.3 Division

The following code is a result of the simple division call made just before the return statement. The C++ code can be seen here:

```
     num = num / 10;
```

The corresponding assembly code:

```
mov     eax, DWORD PTR [rbp-8]
        movsx  rdx, eax     \\rdx contains num
        imul   rdx, rdx, 1717986919 \\similar pattern as modulo
        shr    rdx, 32
        mov    ecx, edx
        sar    ecx, 2
        cdq          \\doubles the size of eax
        mov    eax, ecx
        sub    eax, edx
        mov    DWORD PTR [rbp-8], eax
```

The division operator uses a similar algorithm to that of the modulo. The 3 function calls shr, mov and sar are called in a similar way to modulo. Interestingly, the command cdq is called after. The function call cdq can be used to produce a quadword dividend from a doubleword before doubleword division. In this case, it doubles the size of eax. The last three lines of assembly are also comparable to those in the modulo. The final result is stored in the original location as this is a re-declaration of num.

Combining the features previously discussed will result in assembly that is essentially a combination of the previous functions. With the capabilities discussed so far, one can created many basic functions of a programming language. With that in mind, in order to have a fully operable palindrome.cc, some other basic functions must be discussed. These functions will allow for printing and important statements as well as some other utilities. Below is the full palindrom.cc program along with the corresponding Assembly level code. The full assembly code is available on bottom of this page:

Palindrome.cc

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

int main()
{
    int n, num, digit, rev = 0;
    cout << "Enter a positive number: ";
    cin >> num;
    n = num;
    do
    {
        digit = num % 10;
        rev = (rev * 10) + digit;
        num = num / 10;
    } while (num != 0);
    cout << " The reverse of the number is: " << rev << endl;
    if (n == rev)
        cout << " The number is a palindrome.";
    else
        cout << " The number is not a palindrome.";
   return 0;
}
```

One unique feature of this assembly code is that the strings are each declared in separate labels at the top of the file. This allows them to be referred to later. This can be seen in this section of the code:

```
LC0:
        .string "Enter a positive number: "
.LC1:
        .string " The reverse of the number is: "
.LC2:
        .string " The number is a palindrome."
.LC3:
        .string " The number is not a palindrome."
```

A second feature which needs to be discussed is the use of the print statement. The first print statement corresponds to the following three lines of assembly code:

```
        mov     esi, OFFSET FLAT:.LC0
        mov     edi, OFFSET FLAT:_ZSt4cout
```

```
call    std::basic_ostream<char, std::char_traits<char> >& std::operator<<
    <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char
    const*)
```

In the first mov statement, the string stored at label .LC0 is stored into register esi. The second mov statement corresponds to utilities that are needed before a function call. Lastly, The call instruction is used to call a function. The call instruction pushes the address immediately after itself on the stack. This will allow the computer to print the string stored at .LC0 while not loosing its place on the stack. The program palindrome.cc produces the following assembly code:

Full Assembly Code:

```
LC0:
        .string "Enter a positive number: "
.LC1:
        .string " The reverse of the number is: "
.LC2:
        .string " The number is a palindrome."
.LC3:
        .string " The number is not a palindrome."

main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 0
        mov     esi, OFFSET FLAT:.LC0
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >& std::operator<<
            <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char
            const*)
        lea     rax, [rbp-16]
        mov     rsi, rax
        mov     edi, OFFSET FLAT:_ZSt3cin
        call    std::basic_istream<char, std::char_traits<char> >::operator>>(int&)
        mov     eax, DWORD PTR [rbp-16]
        mov     DWORD PTR [rbp-8], eax
.L2:
        mov     edx, DWORD PTR [rbp-16]
        movsx   rax, edx
        imul    rax, rax, 1717986919
        shr     rax, 32
        sar     eax, 2
        mov     ecx, edx
        sar     ecx, 31
        sub     eax, ecx
        mov     DWORD PTR [rbp-12], eax
        mov     ecx, DWORD PTR [rbp-12]
        mov     eax, ecx
        sal     eax, 2
        add     eax, ecx
        add     eax, eax
        sub     edx, eax
        mov     DWORD PTR [rbp-12], edx
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, edx
```

```
        sal     eax, 2
        add     eax, edx
        add     eax, eax
        mov     edx, eax
        mov     eax, DWORD PTR [rbp-12]
        add     eax, edx
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-16]
        movsx   rdx, eax
        imul    rdx, rdx, 1717986919
        shr     rdx, 32
        mov     ecx, edx
        sar     ecx, 2
        cdq
        mov     eax, ecx
        sub     eax, edx
        mov     DWORD PTR [rbp-16], eax
        mov     eax, DWORD PTR [rbp-16]
        test    eax, eax
        jne     .L2
        mov     esi, OFFSET FLAT:.LC1
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >& std::operator<<
            <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char
            const*)
        mov     rdx, rax
        mov     eax, DWORD PTR [rbp-4]
        mov     esi, eax
        mov     rdi, rdx
        call    std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
        mov     esi, OFFSET FLAT:_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
        mov     rdi, rax
        call    std::basic_ostream<char, std::char_traits<char>
            >::operator<<(std::basic_ostream<char, std::char_traits<char> >&
            (*)(std::basic_ostream<char, std::char_traits<char> >&))
        mov     eax, DWORD PTR [rbp-8]
        cmp     eax, DWORD PTR [rbp-4]
        jne     .L3
        mov     esi, OFFSET FLAT:.LC2
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >& std::operator<<
            <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char
            const*)
        jmp     .L4
.L3:
        mov     esi, OFFSET FLAT:.LC3
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >& std::operator<<
            <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char
            const*)
.L4:
        mov     eax, 0
        leave
        ret
-------------------------Generated by Import Statements-------------------------------------
__static_initialization_and_destruction_0(int, int):
```

```
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-4], 1
        jne     .L8
        cmp     DWORD PTR [rbp-8], 65535
        jne     .L8
        mov     edi, OFFSET FLAT:_ZStL8__ioinit
        call    std::ios_base::Init::Init() [complete object constructor]
        mov     edx, OFFSET FLAT:__dso_handle
        mov     esi, OFFSET FLAT:_ZStL8__ioinit
        mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
.L8:
        nop
        leave
        ret
_GLOBAL__sub_I_main:
        push    rbp
        mov     rbp, rsp
        mov     esi, 65535
        mov     edi, 1
        call    __static_initialization_and_destruction_0(int, int)
        pop     rbp
        ret
```

Lastly, the code generated below the green dashed comment is what is generated by the two import statements. The code consists of several labels which serve as utilities for basics functionality of those libraries.

4 Conclusions

If there is one idea to take away from this course it is to be thankful that there are compilers which do much of the complicated pattern matching work for the user. It is interesting to look back on programming languages like python and see how intuitive they are. All languages must be compiled down to assembly and down to bite code. When attempting to figure out assembly code, it was important to edit the source code down to more reasonable sub functions, with that in mind there are some common patterns in Assembly code which are somewhat intuitive.

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

[oracle] oracle link

[aldeid] aldeid