Как работает код:

# Разбиение на блоки

Первый этап это разбитие на блоки, которые дальше будут преобразованы в токены, их запись в вектор

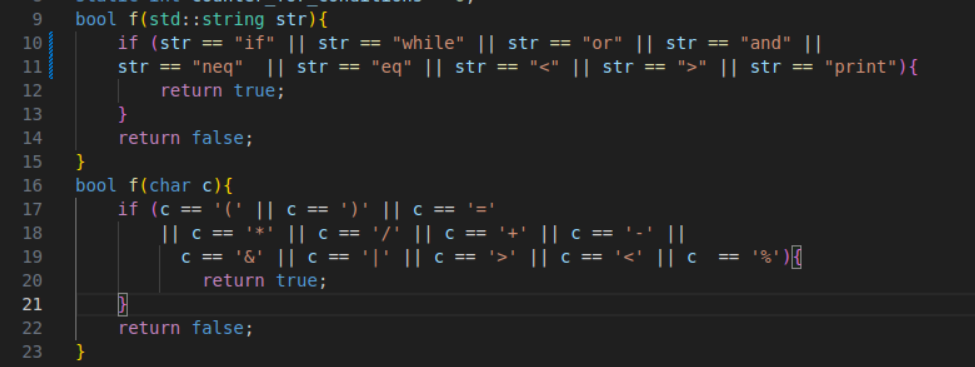
Этот блок обозначен как breaking into block



Мы посимвольно читаем из файла чтобы если он столкнулся с каким-нибудь ключевым словом или оператором\* или круглой скобкой, то он смог его различить, несмотря на то что нет пробелов.

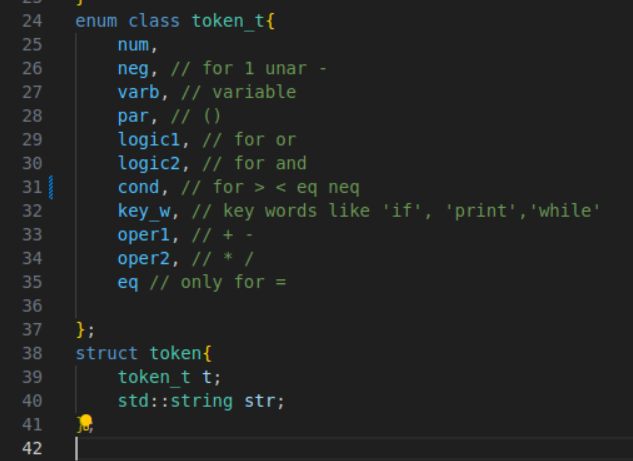
Это выполняет функция f( в этом коде только с char)

\* дальше под оператором будет еще и пониматься круглая скобка



Если он встречает пробел ( s == 32) то добавляет в одномерный вектор branch<std::string> строку, которая составляет блок, если это строка не пустая. Дальше если символ не пробел он смотрит операнд ли это. Если да то он вначале добавляет слово которое было до него( конечно же если оно не пустое) дальше добавляет сам оператор. Строка str нужна для записи для ключевых слов или переменных, которые могут занимать больше 1 символа. Также есть особые символы ; , { , } они особенные потому что эти символы делят полноценные строки(branch), который может обработать синтаксическое дерево и дальше обработчик дерева. Tree это двумерный массив из строк состоящий из блоков.

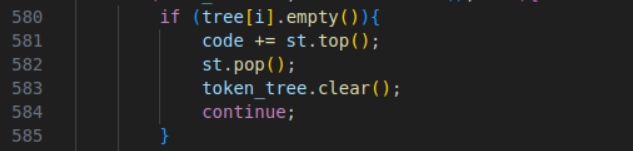
# Токенезация



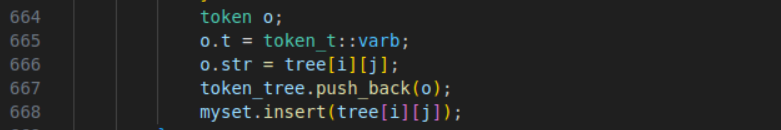
Вся структура токена и типа токена написано в комментариях.

дальше идет обход всего вектора tree в котором блоки.

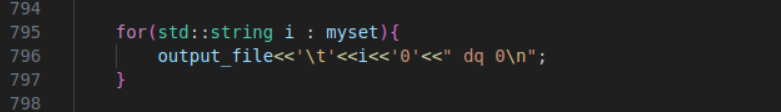
Там простой if который разделяет на блоки на разные типы токенов, и если он не нашел никакого подходящего ключевого слова, то он присваивает тип переменной, что стоит отметить это то, что чтобы отличить числа от переменной, он проверяет первый символ является ли цифрой, то он присваивает тип числа



Это кусок кода объясню позже, когда буду рассказывать про циклы или условия. Также что еще стоит отметить это то, что для минуса есть отдельная проверка. Если он перед открывающийся скобкой ( или перед какой-то операцией или перед знаком равно, то это унарный минус, это отдельный тип. Еще чтобы сохранять название переменной, чтобы потом его добавить в статическую память ( section .data) я добавляю их названия в set myset



, а потом добавляет их по очереди в файл.



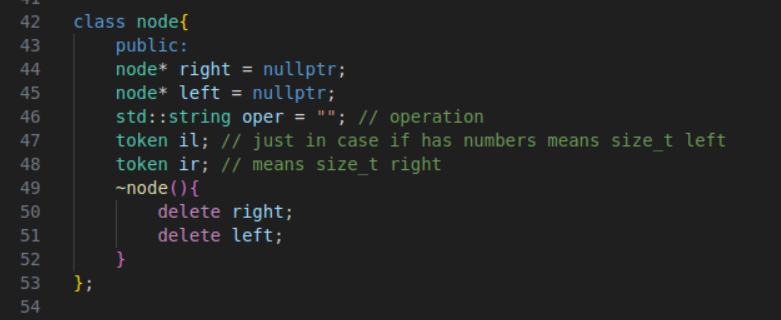
Стоит добавить, что вся токенезация нужна для правильного выстраивания порядка операций.

# Обработка строки( построения синтаксич-

# еского дерева)

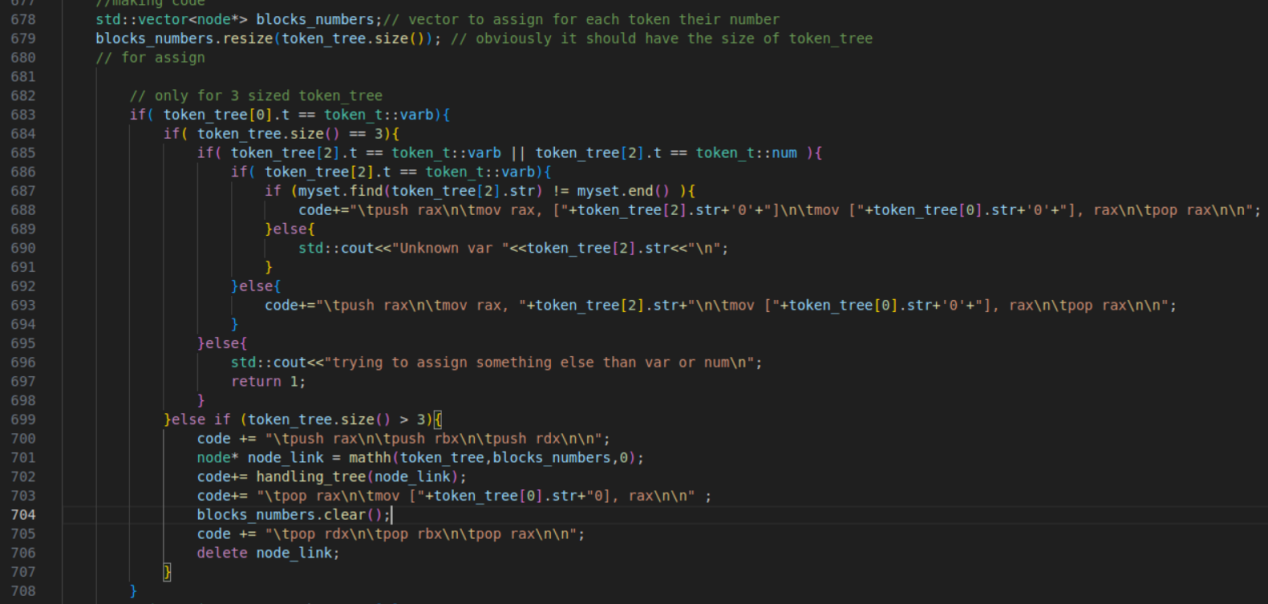
Комментарий making code.

Стоит вначале объяснить структуру дерева.



Так как дерево будет создаваться в функции, которая рекурсивная, то она будет сохраняться в динамической памяти. И чтобы не произошло memory leakage, то стоит написать свой деструктор.

Чтобы не использовать полиморфизм, так как он не очень эффективный, я сохраняю излишнюю информацию, так как правым и левым концом узла может быть и токен или другой узел.



Если первый токен это переменная, то он проверяет размер строки. Если 3, то это значит, что структура строки выглядит так varb = varb/num и мы просто присваиваем переменной1 значение переменной2. также идет проверка на существование переменной. Также в название переменной в output файл добавляется 0, чтобы убрать возможность случайно или нет занять ключевое слово в ассемблере.

Если количество токенов больше, вызывается функция (строчка 701) для обработки математических и логических выражений, которое строит синтаксическое дерево, функция mathh.



Стоит объяснить параметры функции.



Block\_numbers<node\*> (b) это вектор из ссылок на узлы. Это нужно, чтобы если с каким-то числом уже проведена операция до этого, то нынешняя операция проводилась уже с результатом этой операции. Также чтобы случайно не обработать одну и ту же операцию 2+ раз. В коде это реализовано так, что вместо token в node будет сохраняться ссылка на другой узел.

Как это работает. Допустим у нас есть строка x = 10+ 3\*9;

В начале block\_numbers будет так

nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr

X = 10 + 3 \* 9

Первая операция будет 3 \* 9. структура узла будет

left = nullptr

right = nullptr

il = (token\_t::num, “3”)

Ir = (token\_t::num, “9)

oper = “\*”;

Допустим что ссылка на этот узел будет называться link1

Тогда block\_numbers будет

nullptr, nullptr, nullptr, nullptr, link1, link1 , link1

X = 10 + 3 \* 9

Дальше будет операция +

И он за левое значение возьмет 10 так как под его индексом в block\_numbers значение nullptr. А за правое значние он возьмет ссылку link1. Структура этого узла будет выглядеть так .

left = nullptr

right = link1;

il = (token\_t::num, “10”)

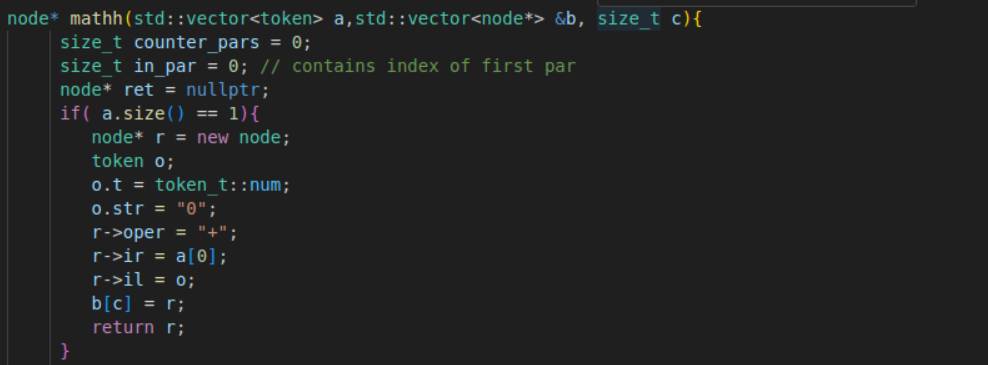
Ir = null

oper = “+”;

Вот зачем нужен block\_numbers и его размер совпадает с размером строки. Также этот вектор я передаю по ссылке

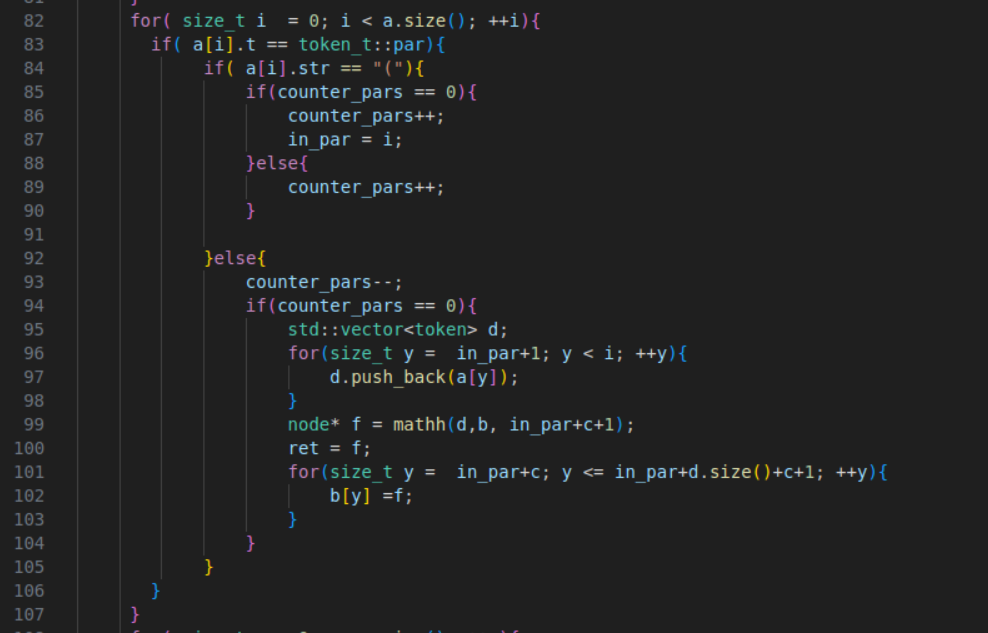
Поэтому этот вектор будет работать в рекурсивной функции

Вектор a состоящий из токенов это просто строка которую нужно обработать. А вот число c служит для определённой цели. Я уже говорил что функция рекурсивная, но его рекурсия заключается в том, что он то что для скобок он вызывает себя и записывает результат ( так результат mathh это сслыка на узел) в block\_numbers под индексами всех токенов внутри скобок и самих скобок. И чтобы эти индексы не перепутать , потому что вектор a<token> может уменьшиться, есть число c, в которое мы записываем смещение. Чтобы записать смещение в функции есть

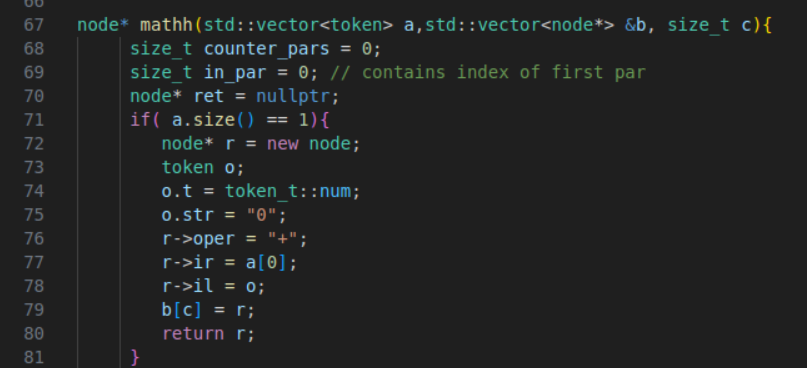


In\_par который хранит индекс первой открывающей скобки, то есть смещение.

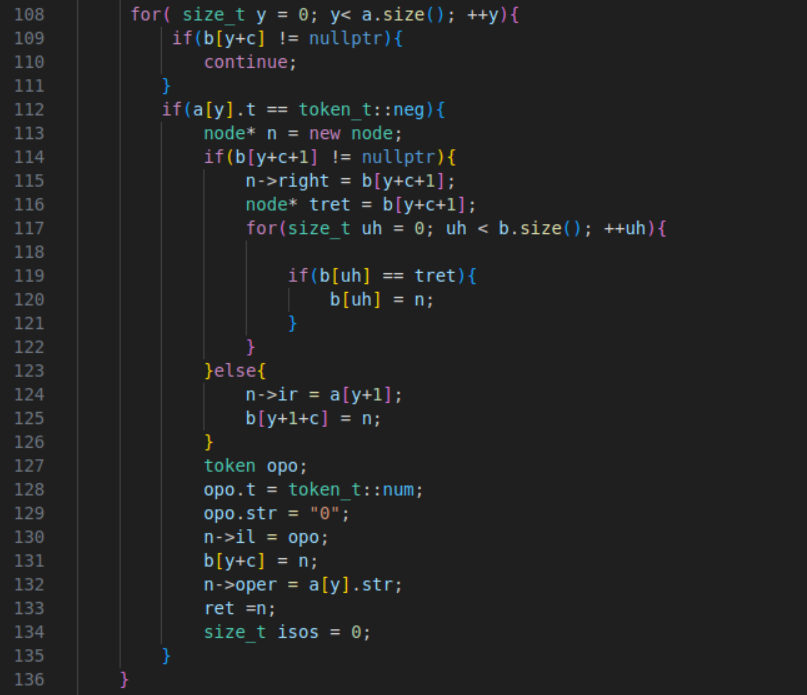
Сама обработка скобок выглядит так,



Мы проходим весь массив a<token> и ищем скобки. Если мы нашли открывающую скобку и conter\_pars( это число равно нулю, когда на все открывающийся скобки есть закрывающая) равен 0( то есть это первая открывающая скобка) то мы записываем в in\_par его индекс ну и прибавляем 1 к counter\_pars. Если скобка закрывающая мы уменьшаем counter\_pars на 1 и проверяем, чтобы на каждую открывающую скобку была закрывающая. Если это так, то мы создаем новый d<token> который является так сказать подвектором а<token> ( начиная со с первого токена после скобки (in\_par+1) и до закрывающей скобки ( i то есть нынешний токен)) и будет уже новым a<token> в новой функции mathh. В node \*f мы записываем результат обработки этой строки. Что стоит подметить, так это то, что сдвиг в новой функции mathh будет не in\_par +1, in\_par + c +1. Не стоит забывать, что скобок может быть больше , чем 2 штуки. Поэтому мы прибавляем еще и сдвиг нынешней функции mathh. Дальше мы просто записываем результат node\*в вектор d aka block\_numbers включая скобки. Перейдём к обработке других случаев, если вся мат выражение a<token> размером 1( строчка 71), то это может значить только одно, что кто-то взял обычное число в скобку, и чтобы обрабатывать это я представляю это как сложения числа или переменной c 0 и возвращаю ссылку на эту операцию.

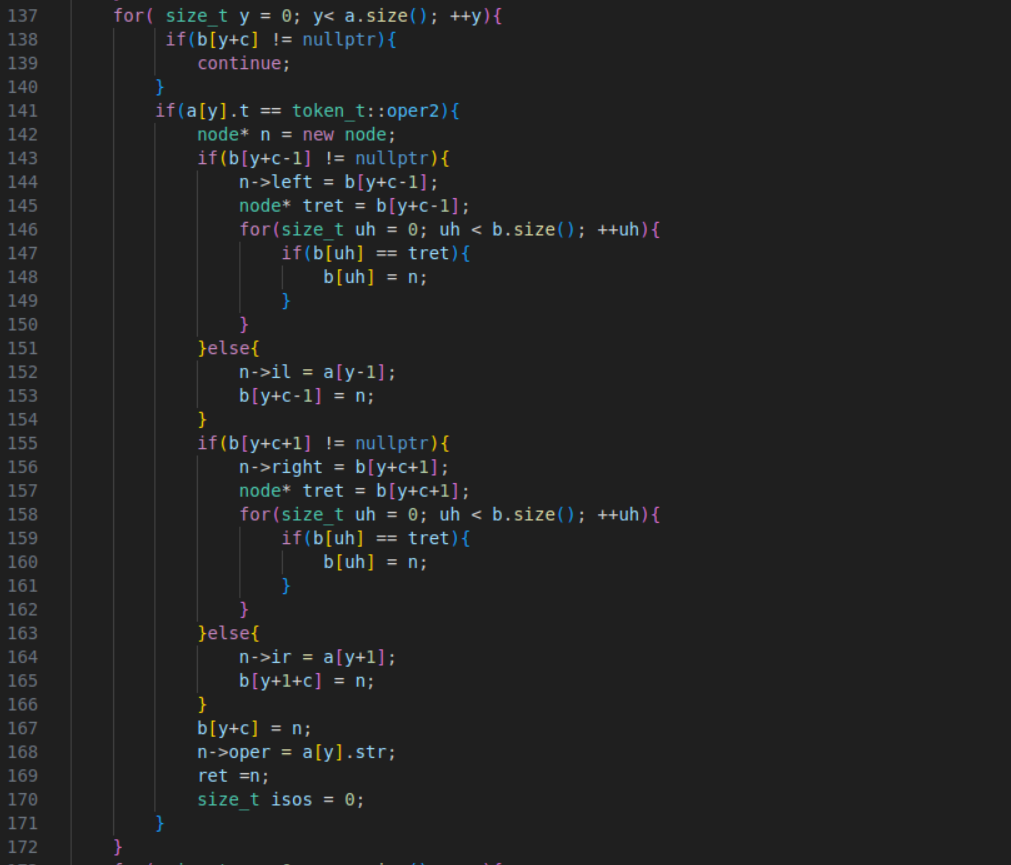


Дальше обработка унарного минуса который имеет тип token\_t::neg он идет по приоритету операций сразу после скобок.



Вначале пропускаю все что уже было обработано(строка 109) ( e.g. то есть под его индексом + смещение уже что-то написано в b aka block\_numbers). Дальше мы проверяем, что мы должны сделать минус, некоторое выражение (строчка 112) или обычный токен. Если под индексом токена находящегося справа от унарного минуса есть выражение в b aka block\_numbers,то имеем дело с выражением. По сути выражение унарного минуса представлено в виде вычитание из нуля числа, переменной или выражения. Конечно же не забываем поменять значения в b aka block\_numbers на новую ссылку( строчка 117), чтобы другие операции обращались не к ссылкам на старые выражения, а уже на ссылку нового выражения.

Дальше идет одинаковый код для всех остальных типов операций и рассматривать их для каждого типа не имеет смысла, так как писался один тот же код, чтобы был правильный приоритет операций. То есть мы ищем какой-то определённый тип токенов по очереди и применяем один и тот же код для построения дерева. Для примера я взял token\_t::oper2 то есть \* / %.

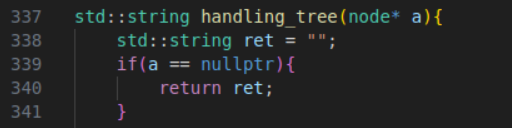
  
Вначале идёт стандартный пропуск всех обработанных токенов ( строчка 138). Дальше мы проверяем обработали ли левый операнд( строчка 143). Если да то мы присваиваем node\* n n->left значение последней операции проведённой над этим операндом. Дальше стандартное изменение всех ссылок выражения левого операнда на новую ссылку новой операции n. Тоже самое проводим и с правым операндом. Дальше мы присваиваем самой операции новую ссылку. Добавляем собственно саму операцию, а точнее ее токен. Переменная isos нигде не используются и была нужна чтобы было удобнее работать с дебагером и ставить на эту строку метку. Все точно также проводим и для остальных операций, только вместо token\_t::oper2 будут другие типы. Мой приоритет операций

1. Скобки // par
2. Унарный минус // neg
3. \*, /, % // oper2
4. +, - // oper1
5. eq, neq, >, < // cond
6. and // logic2
7. or // logic1

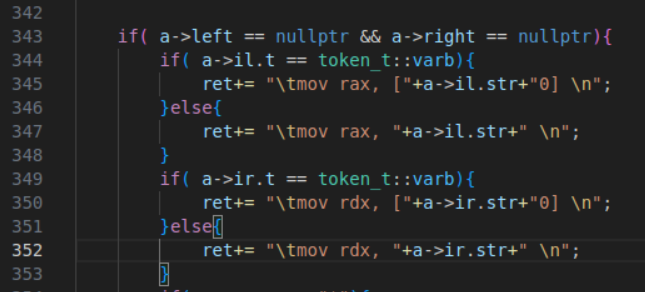
# Обработка дерева

Комментарий handling tree

Так выглядит эта рекурсивная функция по обработке синтаксического дерева

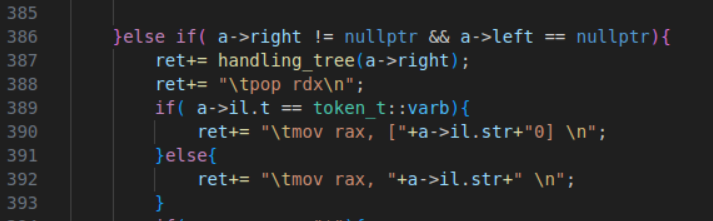


Как работает эта функция, во-первых она проверяет, что ее вызывают не на ссылку с несуществующим узлом. Если это так то возвращается пустая строка. Дальше мы делим обработку на 4 случая. Когда левый и правый просто токены. (строчка 343)

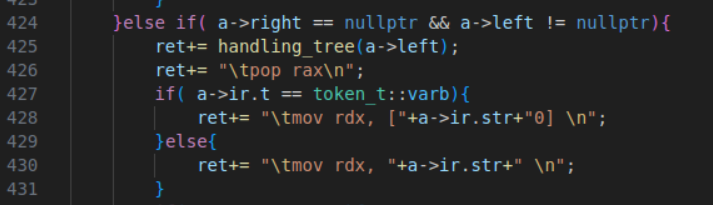


Это нужно, чтобы мы знали, откуда доставать данные для помещения их в регистры rax, и rdx. Очевидно, что если справа и слева токены, то мы просто помещаем значения переменной или числа в регистры( конечно же еще проверяем, что мы пихаем. Так как для обработки переменных нужен специальный алгоритм.

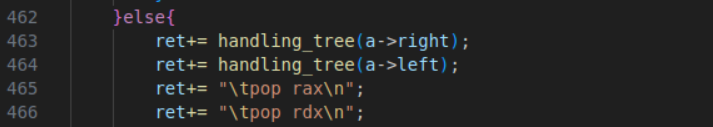
Когда слева токен а справа узел



Если справа узел, то мы вызываем рекурсивно эту функцию, и достаем значение из стека. Так как сама функция Handling tree по сути пишет ассемблер код. Мы записываем все действия которые она написала к нынешнему возвращаемому значению ret. Ассемблер Код написан так, чтобы результат пизался в стек(e.g. то есть результат выполнения правого узла добавляется в стек, откуда мы и его вынимаем в регистр rdx).

Когда справа токен а слева узел  


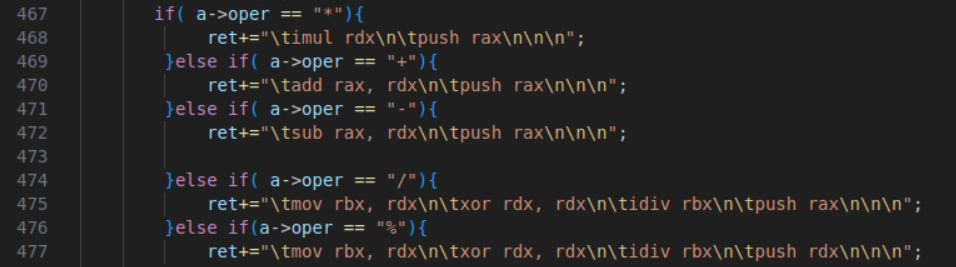
Практически тоже самое как и в предыдущем случае.

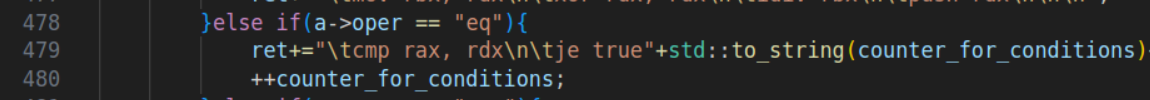
Когда оба узлы  
  


Пишу else так как больше случаев не может быть и нету смысла делать лишнюю проверку.

Работает по принципу предыдущих двух. Что стоит отметить, так это то, что я начинаю идти по дереву с максимально правого конца и двигаюсь дальше.   
к

Теперь перейдем к обработке логических и математических выражений. Так как все 4 случая только определяют откуда мы берем значения для регистров, то очевидно, что операции по работе с этими двумя регистрами одинаковые и выполнение над ними операций одинаковые.  
  
Вначале арифметические операции + - \* / %

  
  
Обработка плюс и минус и умножить стандартная и ее результаты мы пушим в стек. На что стоит обратить внимание так это / %  
  
Так как нужны определенные регистры я присваиваю его значение rbx зануляю rdx. Дальше в зависимости от того, что мне нужно я пушу в стек разные регистры. Если целая часть то rax, если остаток то rdx.

(напоминаю что a это ссылка на узел. А его значение oper хранит операцию).  
  
Теперь переходим к обработке логических выражений.  
  
Строки 478 - 496   
 первый это обработка eq то есть ==   
  


Вначале стоит объяснить что такое counter\_for\_conditions. Это счётчик для условий. Так как возникает необхожимость создавать метки, и чтобы они были уникальными заводитяс счетчик  
сам ассемблер код для eq выглядит так (для первого условия, так как там true0 и end0  
  
  
  
  
  
  
  
  
  
  
  
  
  
Если правое и левое значения равны то он пушит в стек 1 иначе 0

cmp rax, rdx

je true0

push 0   
 jmp end0

true0:

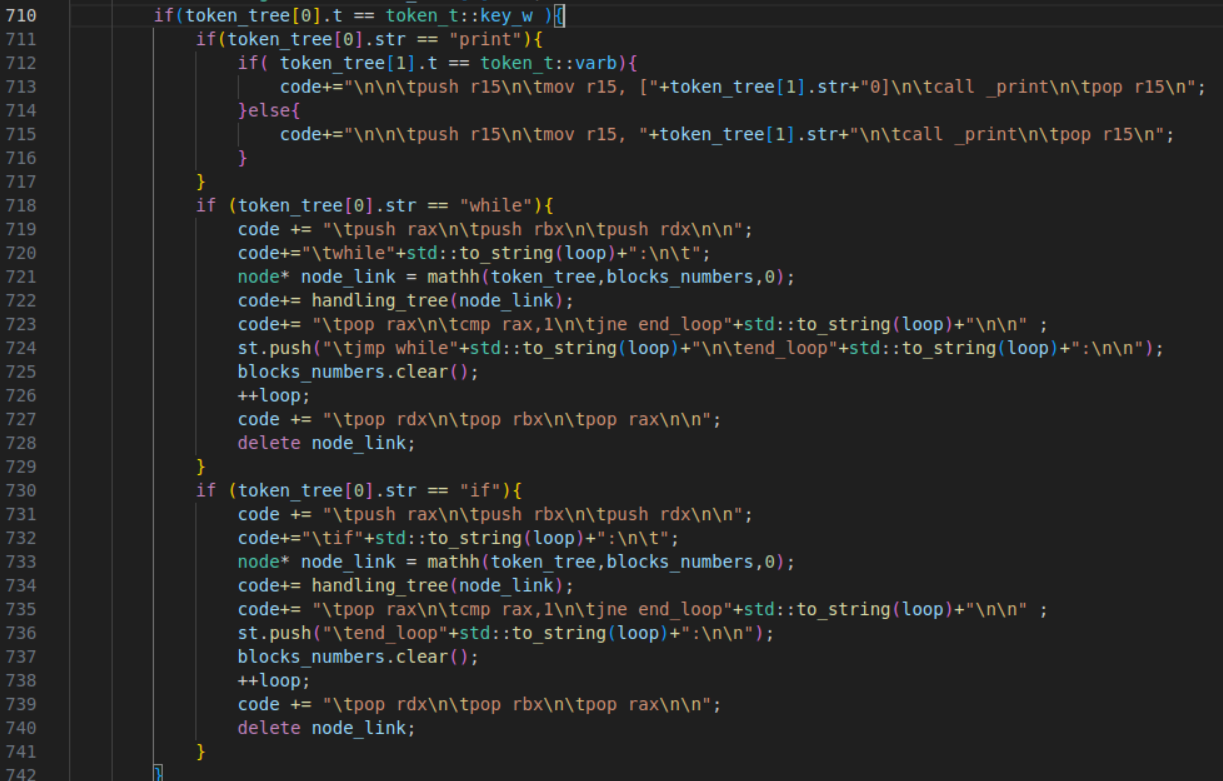
push 1

end0

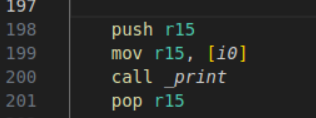
Для neq ( !=) вместо je стоит jne.

Для больше jg для меньше jl а для and и or вместо cmp провожу and и or соотвественно над 2 регистрами и пишу jnz все остальное одинаковое.

# Обработка while и if и print



Начнем с print. В самом начале файла .asm я пишу

Extern \_print и на процессе линковки конечного исполняемого файла добавляю эту процедуру, которая написана в отдельном файле print.asm. Выглядит это так.  


Сохраняю значение регистра r15, пишу туда нужное значение и вызываю саму функцию.

В с++ я просто вставляю, то что мне нужно вместо [i0]

Перейдем к while. Переменная loop счетчик для меток if и while. Как выглядит структура while( стоит учитывать, что я пишу для первого while и поэтому счетчик равен 0)  
  
while0:

Выполнение всех лог операцийи пуш в стек результата  
pop rax

cmp rax,1

jne end\_loop0

---------------------------

Какие-то команды

-----------------------------

jmp while0

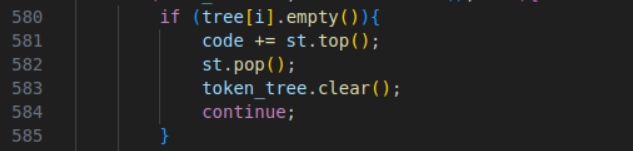
end\_loop0:

Как работает: если мы встречаем while то мы добавляем все нужные метки и запихиваем в стек(st<std::string>) конец наших меток

jmp while0

end\_loop0:

И так как } тоже является разделителем строки и так как она стоит после ;то это означает, что она создает пустую строку. Теперь возвращаемся к началу когда происходила токенезация.



Когда мы встречаем пустую строку, то в строку кода файла (code) мы просто добавляем конец нашего while. Это нужно, чтобы можно было обрабатывать вложенные циклы, а еще это делает работу удобнее так как, мы работаем только со строками программы.  
Для if все тоже самое, только в конце не вставляется jmp if0