

Компилятор за выходные: избавляемся от переменных

haqreu

Компилятор за выходные: избавляемся от переменных

Средний

15 мин

13K

Вопрос из области ненормального программирования: насколько сложные программы вы сможете написать на питоне, не пользуясь в принципе переменными (а также аргументами функций), за исключением пары глобальных массивов? Правильный ответ: да любой сложности. Если что-то можно сделать на ассемблере, то уж на питоне и подавно! Правда, пусть лучше код вместо меня сгенерирует машина :)

Продолжаем разговор о минималистичном компиляторе, который вполне реально написать за выходные. Задачей стоит транслировать код из придуманного мной языка в x86 ассемблер. [Мой компилятор состоит](#) из 611 строк кода, при этом не имеет ни единой зависимости:

```
ssloy@khronos:~/tinycompiler$ cat *.py|wc -l
611
```

Несмотря на то, что целевым языком является ассемблер, я не мазохист, и пришёл к этому постепенно. Сначала я транслировал код в питон, и постепенно урезал функционал целевого языка, пока не остался голый ассемблер. Тема сегодняшнего разговора: генерация кода на питоне без использования переменных.

Оглавление

[Синтаксические деревья и наивный транслятор в питон](#)

[Лексер/парсер](#)

2'. [Проклятый огонь, или магия препроцессора C](#)

[Таблицы символов: области видимости переменных и проверка типов](#)

[Стек и транслятор в питон без использования питоновских переменных \(эта статья\)](#)

Транслятор в ассемблер

Рейтрейсинг :)

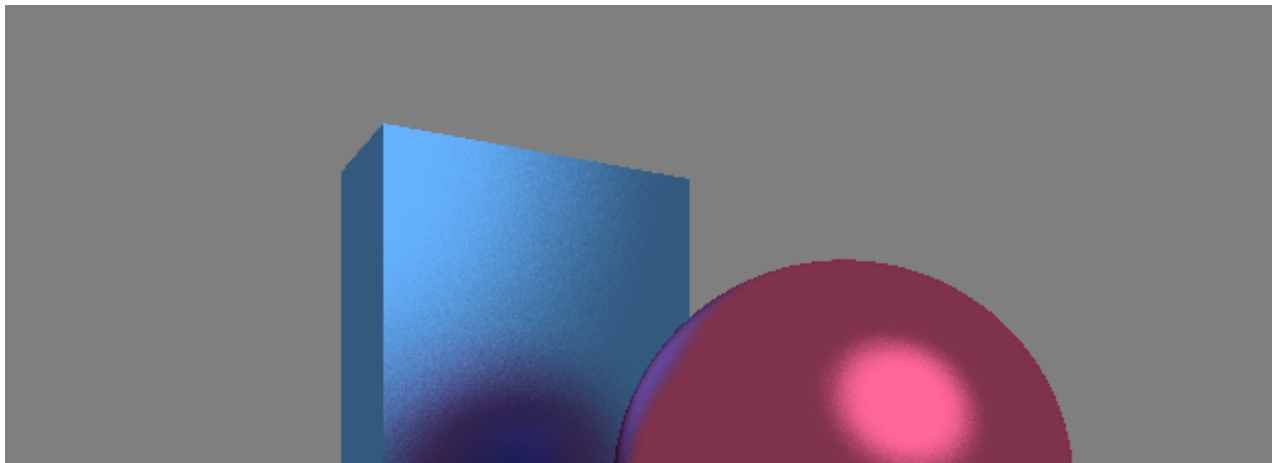
Избавляемся от зависимостей: пишем лексер и парсер сами

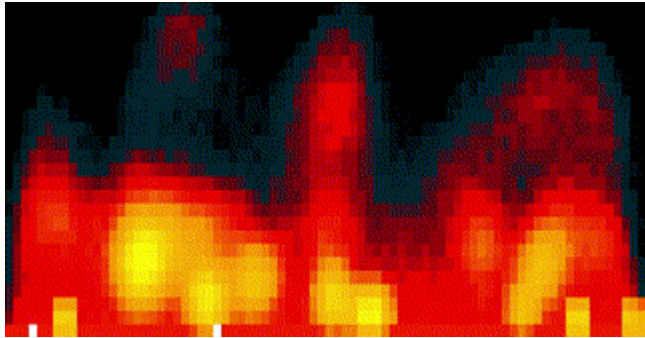
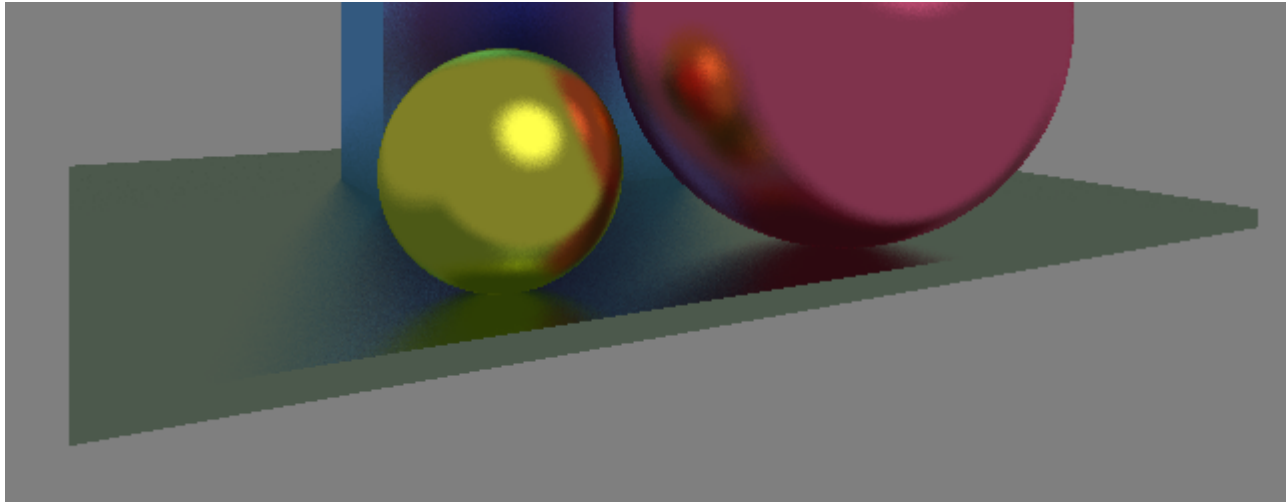
Оптимизирующий компилятор?

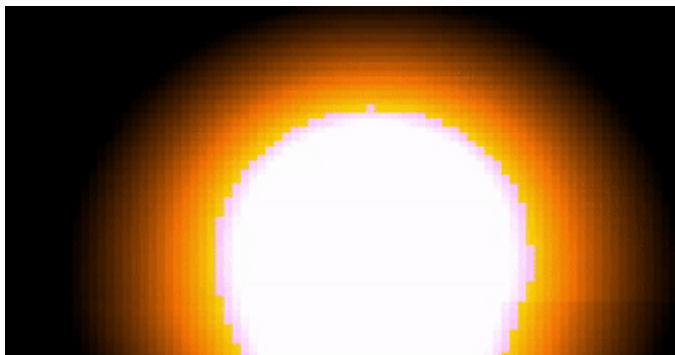
Сборщик мусора?

GFX!

Мой язык крайне примитивен, но при этом позволяет создавать вполне полноценные программы, вот результат работы нескольких программ, которые я написал специально для этого курса:







Код для этих примеров [лежит тут](#). Я уже [подробно описывал](#) программу отрисовки огня (и заодно программирование с использованием только лексера), остальные примеры на очереди.

Знаю, что у меня профдеформация в сторону компьютерной графики, но я терпеть не могу примеры вроде вычисления чисел Фибоначчи и всяких факториалов (буквально следующий пример в данной статье - это факториал :)). Я хочу, чтобы мои студенты могли проверять свои компиляторы на чём-то приятном для взора.

Рекурсия для самых маленьких

Факториал целого положительного числа n - это произведение всех положительных целых чисел, меньших или равных n . Например, факториал числа 7 равен $7*6*5*4*3*2*1$, что равняется 5040. Факториал числа n записывается как $n!$, и задача его вычисления сводится очевидным образом к вычислению $(n-1)!$, а именно, $n! = n * (n - 1)!$

Сама собой напрашивается рекурсивная имплементация:

```
ssloy@khronos:~$ python3 <<<'
def factorial(n):
    result = 1
    if n > 1:
        result = n * factorial(n-1)
    return result
print(factorial(7))
'
```

5040

С подобного кода зачастую начинают знакомство с программированием, но обычно оставляют за скобками объяснение того, как непосредственно машина его выполняет. Основным инструментом в таком коде является вызов функции, но как он работает?

Типовой (не единственно возможный, именно типовой) механизм реализации вызова функции основан на сохранении аргументов и локальных переменных функции на стеке и выглядит следующим образом:

В точке вызова на стек помещаются параметры, передаваемые функции (+ обычно номер инструкции, на которую нужно вернуться после завершения работы функции).

Вызываемая функция в ходе работы кладёт на стек собственные локальные переменные.

По завершении вычислений функция очищает стек от своих локальных переменных, записывает результат (обычно в один из регистров процессора).

Команда возврата из функции считывает из стека адрес возврата и выполняет переход по этому адресу. Либо непосредственно перед, либо сразу после возврата из функции стек очищается от аргументов функции.

Нетрудно видеть, что необходимость расширения стека диктуется требованием восстановления состояния вызывающего экземпляра функции (то есть её параметров, локальных данных и адреса возврата) после завершения вызванной функции.

Таким образом, при каждом рекурсивном вызове функции создаётся новый набор её параметров и локальных переменных, который вместе с адресом возврата размещается на стеке. Чтобы не быть голословным и не теоретизировать лишку, давайте добавим одну строчку интроспекции в наш код с факториалом:

```
ssloy@khronos:~$ python3 <<<'
import inspect
def factorial(n):
    result = 1
    # вот эту строчку я добавил:
    print("instances of n:", [frame[0].f_locals["n"] for frame in reversed(inspect.stack()[::-1])])
    if n > 1:
        result = n * factorial(n-1)
    return result

print(factorial(7))
'
```

instances of n: [7]
instances of n: [7, 6]
instances of n: [7, 6, 5]
instances of n: [7, 6, 5, 4]
instances of n: [7, 6, 5, 4, 3]
instances of n: [7, 6, 5, 4, 3, 2]
instances of n: [7, 6, 5, 4, 3, 2, 1]
5040

При каждом вызове `factorial(n)` добавленная мной строчка проходит по стеку, и выводит на экран все экземпляры переменной `n`, что находит. Для вычисления $7!$ функция `factorial(n)` вызывается 7 раз, и нетрудно видеть, что в какой-то момент машина хранит все 7 экземпляров переменной. Разумеется, для переменной `result` ровно такая же картина.

Основная мысль этой статьи

Питон делает за нас массу работы, но это не значит, что без его помощи нам не обойтись. Ассемблер не даёт роскоши автоматической передачи аргументов, надо руками работать со стеком. Я предлагаю по-прежнему использовать питон в качестве целевого языка, но урезать используемые возможности до уровня ассемблера: по факту мы будем писать на ассемблере, но с синтаксисом питона. У такого подхода два преимущества:

Мы не обязаны строго следовать всем ограничениям сразу. Например, я могу ограничить возможности передачи аргументов функциям, но не обязан сразу прыгать в мир с четырьмя регистрами. Я вполне могу пользоваться питоновским разборщиком выражений, написав комфортное мне `5//3`, если такое потребуется, вместо километровой портянки инициализации всех нужных регистров с требуемыми флагами. Таким образом, я могу в любой момент иметь работающий компилятор, и допиливать его по кусочкам.

Кроме того, мы не обязаны прыгать с головой в не очень дружелюбный для новичков мир, в котором зачастую даже примитивный `print()` не работает, а уж как работают дебагеры - это отдельная песня. Мы можем использовать любимый освоенный IDE, ставить точки останова, выводить на экран что угодно и комфортно изменять руками сгенерированный код, чтобы понять, что же в нём пошло не так.

Мы не обязаны пользоваться встроенным стеком питона, можно эмулировать его работу при помощи самописного стека. Давайте заведём глобальный массив `stack`, и будем в нём хранить аргументы и локальные переменные функций. Тогда вычисление факториала можно переписать (руками!) следующим образом:

```
ssloy@khronos:~$ python3 <<<'
def factorial():                                # n is stack[-1]
    stack.append(1)
    if stack[-2] > 1:                            # n is stack[-2], result is stack[-1]
        stack.append(stack[-2]-1)
        stack[-2] = stack[-3] * factorial()      # n is stack[-3], result is stack[-2]
    stack.pop()
    return stack.pop()
```

```
stack = []
stack.append(7)
print(factorial())
stack.pop()
',
5040
```

В этом коде мы больше не пользуемся в принципе переменными питона за исключением одного-единственного массива `stack`. Эта программа очень похожа на то, что я хочу получить автоматически при помощи моего компилятора, и переход от предыдущей к этой и есть основная мысль статьи. Отставьте свою чашку с чаем, посмотрите на код внимательно, если нужно, нарисуйте стек на листочке.

Я написал код руками, и мне не очень нравится, что я был должен вручную отслеживать положение переменных в стеке. Для того, чтобы обратиться к одной и той же переменной `n`, я был вынужден вызывать три разных выражения `stack[-1]`, `stack[-2]` и `stack[-3]`. Это не очень хорошо, нужно найти более дубовый способ. Самое время вспомнить про таблицы символов из прошлого анекдота.

Подопытный кролик

Давайте отложим факториал и рассмотрим простейший нетривиальный пример с несколькими разными вызовами функций. Слева вы видите исходный код на `wend`, справа его трансляция в питон при помощи версии [v0.0.3](#) моего компилятора. Именно она была описана в [прошлой статье про таблицы символов](#). Текущая задача - переписать правую часть руками без использования переменных, практически так же, как я сделал с факториалом.

```
fun main() {
  fun sopfr(n:int):int {
    var div:int;
    fun sopfr_aux(n:int):int {
      var rec:int;
      rec = 0;
      if n % div == 0 {
        rec = div;
        if n!=div {
          rec = rec + sopfr_aux(n/div);
        }
      }
    }
  }
}
```

```
def main():
  def sopfr(n):
    def sopfr_aux(n):
      nonlocal div
      rec = 0
      if n % div == 0:
        rec = div
        if n != div:
          rec = rec + sopfr_aux(n // div)
      else:
        div = div + 1
```



```

    } else {
        div = div + 1;
        rec = sopfr_aux(n);
    }
    return rec;
}

div = 2;
return sopfr_aux(n);
}

println sopfr(42);
}

div = div + 1;
rec = sopfr_aux(n);
return rec;

div = 2;
return sopfr_aux(n);

print(sopfr(42))

main()

```

Если вдруг код на картинке слишком ломает глаза, давайте я его дам прямым текстом:

- wend
- python

Данная программа высчитывает для данного числа сумму его простых множителей. Например, $20 = 5 \cdot 2 \cdot 2$, так что $\text{sopfr}(20) = 2 + 2 + 5 = 9$. Эту функцию часто называют [целочисленным логарифмом](#), что неудивительно, поскольку вполне очевидно, что $\text{sopfr}(a * b) = \text{sopfr}(a) + \text{sopfr}(b)$. Математики довольно активно изучают свойства этой функции, но это несколько выходит за рамки нашего обсуждения :)

В данном коде мне интересно несколько вещей:

У нас есть три вложенных области видимости.

У нас есть функции с разным количеством аргументов и разным количеством локальных функций.

У нас есть обращение к **нелокальной** переменной `div` из функции `sopfr_aux`.

Последнее представляет особый интерес. Давайте вспомним, что в последнем примере с факториалом мне приходилось руками отслеживать положение локальных переменных на стеке, размер которого менялся. Это неприятно, но совсем нестрашно, поскольку все изменения стека внутри одной функции известны во время компиляции: сначала я обращаюсь к переменной `n` как `stack[-1]`, а после добавления локальной переменной `result` стек вырос, и `n` стал предпоследним элементом `stack[-2]`. А что делать, когда стек меняется в рантайме? А ведь это происходит крайне регулярно.

Упражнение 1

Давайте вернёмся к нашему подопытному кролику вычисления `sopfr(n)`. Поставим точку останова на строке `rec = div`:

```
def main():
    def sopfr(n):
        def sopfr_aux(n):
            nonlocal div
            rec = 0
            if n % div == 0:
                rec = div # breakpoint here
                if n != div:
                    rec = rec + sopfr_aux(n // div)
            else:
                div = div + 1
                rec = sopfr_aux(n)
            return rec

        div = 2
        return sopfr_aux(n)

    print(sopfr(42))

main()
```

42 раскладывается на множители 2, 3, 7, то есть, интерпретатор должен пройти через эту строчку трижды. Возьмите листочек и карандаш, и нарисуйте состояние стека в каждый из трёх раз (меня интересуют только переменные, всякие адреса ни к чему).

Давайте я помогу с первой точкой останова. Функция `main()` на стек кладёт 42, и это соответствует переменной `n` в функции `sopfr(n)`. Она, в свою очередь, кладёт 2 на стек (переменная `div`), и затем снова кладёт 42 (аргумент функции `sopfr_aux(n)`). Ну а `sopfr_aux` создаёт локальную переменную `rec`, положив на стек 0, и доходит до точки останова, поскольку 42 делится на 2. Таким образом, когда мы дойдём в первый раз до точки останова, стек будет выглядеть следующим образом: [42,2,42,0]. Вот небольшая анимация процесса:

```
def main():
    def sopfr(n):
        def sopfr_aux(n):
            nonlocal div
            rec = 0
            if n % div == 0:
                rec = div # breakpoint here
                if n != div:
                    rec = rec + sopfr_aux(n // div)
```

```

    else:
        div = div + 1
        rec = sopfr_aux(n)
        return rec

    div = 2
    return sopfr_aux(n)

print(sopfr(42))

main()

```

stack

val
var
context

Теперь же оторвитесь от статьи, и попробуйте нарисовать стек для второй и третьей точки останова. Задача тривиальная, но готов спорить, что многие ошибутся. Если кто ошибся - отписывайтесь в комменты, не стесняйтесь :)

► Ответ

Не удивляйтесь, это совершенно нормальный ответ, вот я добавил самую малость интроспекции кода для проверки результата:

► Проверочный код

Таблицы символов спешат на помощь

Когда я переписывал на ассемблерном питоне факториал, я обращался к переменным просто по сдвигу от вершины стека, и уже даже для локальных переменных этот сдвиг был разным, но хотя бы был известен в compile time. А тут всё сильно иначе. Дойдя до нашей точки останова, мы захотим записать `div` в `rec`: одна переменная локальная, а вторая нелокальная, и она закопана очень глубоко по стеку; при этом глубина неизвестна во время компиляции. С другой стороны... Давайте посмотрим, как выглядит таблица символов для нашей программы:

```

Symbol table:
main
├── sopfr
│   ├── n : int
│   ├── div : int
│   └── sopfr_aux
│       ├── n : int
│       └── rec : int

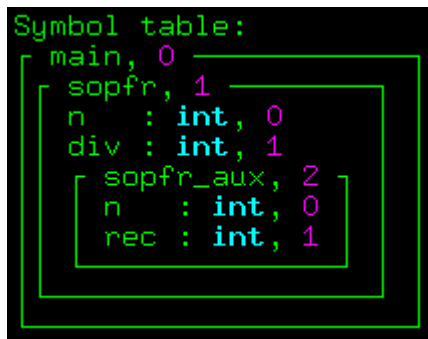
```

У нас есть три вложенных области видимости переменных: `main`, `sopfr`, `sopfr_aux`.

В функции `main` переменных нет, в `sopfr` есть две целочисленные переменные `n` и `div`, а в `sopfr_aux` есть две целочисленные переменные `n` и `rec`.

До сих пор мой компилятор обращался к переменным по их имени, теперь же пора их пронумеровать и про идентификаторы забыть.

[Вот diff коммита](#), в котором я просто добавил счётчики областей видимости и переменных внутри каждой области. По факту, в таблицу символов я добавил вот эти фиолетовые циферки:

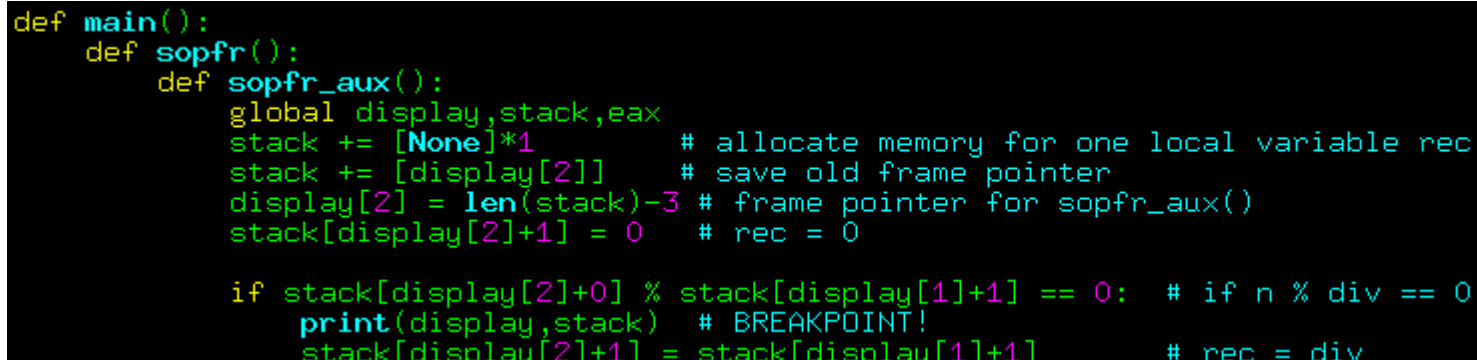


```
Symbol table:
[ main, 0
  [ sopfr, 1
    n : int, 0
    div : int, 1
    [ sopfr_aux, 2
      n : int, 0
      rec : int, 1
    ]
  ]
]
```

Теперь, когда я захочу обратиться к переменной `div`, мне не нужен её идентификатор, мне достаточно знать, что это переменная с индексом 1 в области видимости 1, то есть для идентификации мне хватит пары чисел.

Теперь у нас есть всё необходимое для того, чтобы избавиться от переменных и для `sopfr`, который изрядно сложнее факториала. Напоминаю, что я по-прежнему пишу код руками, мне это нужно для того, чтобы понять, как будет работать мой компилятор.

Вот код картинкой:



```
def main():
    def sopfr():
        def sopfr_aux():
            global display, stack, eax
            stack += [None]*1 # allocate memory for one local variable rec
            stack += [display[2]] # save old frame pointer
            display[2] = len(stack)-3 # frame pointer for sopfr_aux()
            stack[display[2]+1] = 0 # rec = 0

            if stack[display[2]+0] % stack[display[1]+1] == 0: # if n % div == 0
                print(display, stack) # BREAKPOINT!
                stack[display[2]+1] = stack[display[1]+1] # rec = div
```

```

        if stack[display[2]+0] != stack[display[1]+1]: # if n != div
            stack += [stack[display[2]+0]//stack[display[1]+1]] # push n/div ↴
            sopfr_aux() # > sopfr_aux(n/div)
            del stack[-1:] # pop n/div ↵
            stack[display[2]+1] = stack[display[2]+1] + eax
        else:
            stack[display[1]+1] = stack[display[1]+1] + 1
            stack += [stack[display[2]+0]] # push n ↴
            sopfr_aux() # > sopfr_aux(n) call
            del stack[-1:] # pop n ↵
            stack[display[2]+1] = eax

    eax = stack[display[2]+1]
    display[2] = stack.pop() # restore frame pointer
    del stack[-1:] # remove rec from stack

global display,stack,eax
stack += [None]*1 # allocate memory for one local variable div
stack += [display[1]] # save old frame pointer
display[1] = len(stack)-3 # frame pointer for sopfr()
stack[display[1]+1] = 2 # div = 2
stack += [stack[display[1]+0]] # push n ↴
sopfr_aux() # > sopfr_aux(n)
del stack[-1:] # pop n ↵
display[1] = stack.pop() # restore frame pointer
del stack[-1:] # remove div from stack

global display,stack,eax
stack += [None]*0 # no local variables
stack += [display[0]] # save old frame pointer
display[0] = len(stack)-1 # frame pointer for main()
stack += [42] # push 42 ↴
sopfr() # > sopfr(42)
del stack[-1:] # pop 42 ↵
print(eax)
display[0] = stack.pop() # restore frame pointer

display = [ None ]*3 # uninitialized frame pointers
stack = [] # empty stack
eax = None # registers
main()

```

► И он же текстом

На первый взгляд пугающе, но на самом деле ничего сложного. Давайте разберём, что тут происходит. Перво-наперво смотрим на то,

что у нас добавилось глобальных переменных. Теперь у нас есть не только `stack`, но также появились массив `display` и переменная `eax`.

Стек своей роли не менял, переменная `eax` эмулирует регистр процессора, и согласно [договорённости о вызовах функций для x86](#), я буду класть в этот "регистр" возвращаемое значение функций.

А вот с глобальным массивом `display` интереснее. Обратите внимание, что он непустой. Этот массив не будет менять своего размера, количество его ячеек равно количеству разных областей видимости в нашей таблице символов, в данном случае 3. Смысл в том, что при вызове каждой функции мы будем записывать в соответствующую ячейку индекс вершины стека, что позволит нам обратиться к переменной `div` как `stack[display[1]+1]`, а к переменной `rec` как `stack[display[2]+1]`. Эта техника убирает наглухо зависимость от рантайма, каждая переменная всегда идентифицируется одной и той же, постоянной парой чисел, которую мы просто берём из таблицы символов. Разумеется, при вызове функции нам нужно будет сохранить (на стек, куда же ещё) старое значение `display`, а по завершении его восстановить.

Таким образом, тело любой функции с `nlocals` локальных переменных и `nargs` параметров у нас будет выглядеть следующим образом:

```
def foo():
    global display, stack, eax
    stack += [None]*nlocals           # reserve place for local variables
    stack += [display[scope]]         # save old frame pointer
    display[scope] = len(stack)-nlocals-nargs-1 # set new frame pointer for foo()
    ...                               # function body
    display[scope] = stack.pop()       # restore old frame pointer
    eax = ...                         # return value
    if nlocals>0:                     # remove local variables from stack
        del stack[-nlocals:]
```

Ну а вызов такой функции будет выглядеть очень просто:

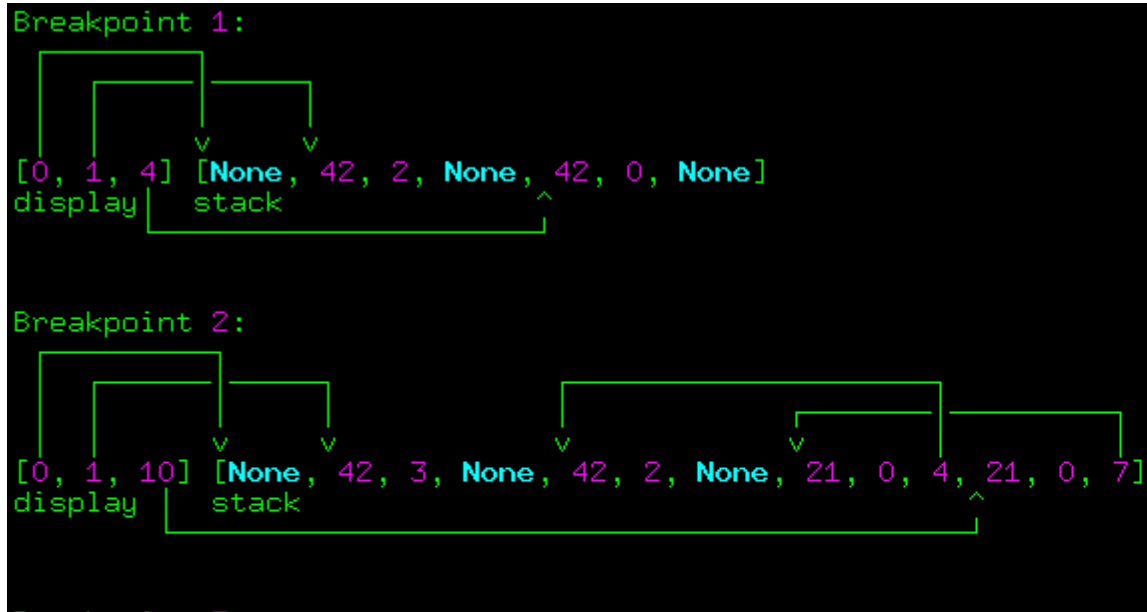
положить на стек `nargs` значений,
вызвать функцию,
удалить со стека `nargs` значений.

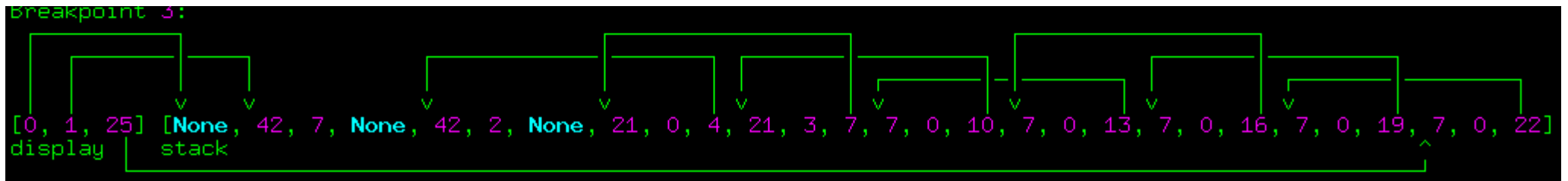
Я не просто так остановился на `sorfi`, я потратил целый день на то, чтобы выбрать пример. Он хорош тем, что у него есть целых три разных функции, у которых разное количество параметров, разное количество локальных переменных, и даже есть нелокальная переменная. Таким образом, у него есть три тела функции, и даже ещё больше разных вызовов, что позволяет увидеть, что оформляются они абсолютно одинаково и исключительно при помощи информации, доступной из таблицы символов.

Упражнение 2

Код мы переписали, но структура-то его не изменилась. Давайте оставим точку останова ровно там же, где и раньше, и в качестве упражнения отрисуем состояние `display` и `stack` все три раза. Сейчас я прятать результат не буду, всё равно сходу стрелочки не запомните. Не смотрите особо на ответ, нарисуйте стек по-честному, оно поможет понять, что и в какой момент мы сохраняем.

Ответ:





Финишная прямая: оценка значений выражений

Осталось совсем немного до ассемблерного питона. Финальный штрих - это оценка значений выражений. Давайте представим, что у нас есть вот такая программа:

```
fun main() {  
    var a:int;  
    var b:int;  
    [...]  
    a+b*2  
    [...]  
}
```

Тогда в нашей таблице символов у нас будет одна область видимости переменных с идентификатором 0, и внутри у неё будут две переменных с идентификаторами 0 и 1, соответственно. В данный момент меня интересует, как посчитать выражение $a+b*2$.

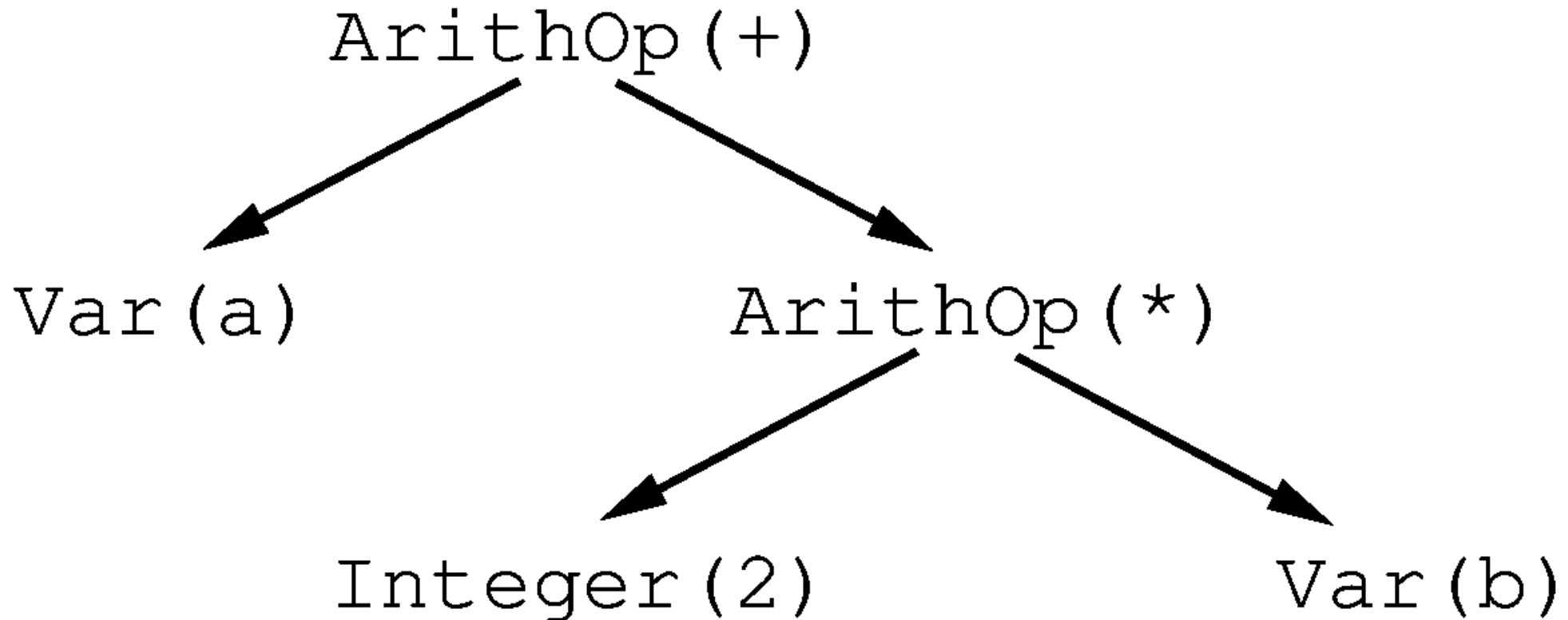
В ассемблере оно будет выглядеть как-то так (напоминаю, что в GNU ассемблере операторы имеют вид `op src dst`):

```
mov display+0, %eax    # найти фрейм функции main  
mov -1(%eax), %ebx     # положить значение b в регистр ebx  
mov -0(%eax), %eax     # положить значение a в регистр eax  
imul 2, %ebx           # ebx = 2 * ebx  
add %ebx, %eax         # eax = eax + ebx
```

Всё бы хорошо, но вот только тут мне пришлось хитрить: для того, чтобы положить `b` в `ebx`, мне нужно испортить значение `eax`, так что

сначала я читал аргумент `b`, и только потом `a`. А ещё я хитро положил результат `2*ebx` назад в `ebx`... А ещё надо помнить, что регистров очень мало, и на выражение `a + b * 2 - foo(c - 3, d + 10*d, bar(a/4))` регистров не хватит гарантированно.

Ну да не беда. Давайте вспомним, что мы пишем компилятор, и что парсер нам построил синтаксическое дерево выражения, которое выглядит следующим образом:



Тогда мы можем сгенерировать код оценки значения выражения при помощи обхода дерева в глубину, используя стек для хранения промежуточных выражений. Представьте, что мы договоримся об очень простой вещи: каждый узел выражения сохраняет своё значение в одном и том же регистре, например, `%eax` (кстати, вызов функции - это тоже выражение, и в примере с `sort` вызов функции именно что сохранял свой результат в `%eax`).

И тогда единственная (рекурсивная) операция оценки выражения может выглядеть следующим образом (для простоты я говорю здесь

только о бинарных операциях типа арифметических):

```
оценить левое выражение (результат сохранён в %eax)
push(%eax)
оценить правое выражение (результат сохранён в %eax)
переложить %eax в %ebx
%eax = pop()
%eax = операция над %eax и %ebx
```

Используя такой подход, выражение $a+2*b$ может быть оценено следующим псевдокодом:

положить a в %eax		\	
push(%eax)			
положить 2 в %eax	\		
push(%eax)			
положить b в %eax		2*b сохранено	a + 2*b сохранено
переложить %eax в %ebx		в %eax	в %eax
%eax = pop()			
%eax = %eax * %ebx	/		
переложить %eax в %ebx			
%eax = pop()			
%eax = %eax + %ebx		/	

Обратите внимание, что код явно неоптимален, очень легко можно сэкономить много чего (собственно, ассемблерный код, который я привёл ранее, сильно проще и не использует стека). Но на данном этапе меня оптимизация не интересует в принципе. Мне нужно найти общий паттерн оценки выражений, и, похоже, он найден! Об оптимизирующем компиляторе мы поговорим в отдельный раз (запланировано).

Зато такой подход имеет неоспоримое преимущество: он использует только два регистра и стек, больше ему ничего не нужно! Так что finishing touch над нашей `sopfr`: все выражения оцениваются в регистр `eax`. Я по-прежнему написал код руками, но опять же,

исключительно для проверки работоспособности подхода. Узрите же питонассемблерный код!

```

        eax = eax + ebx          # rec + sopfr_aux(n/div)
        stack[display[2]+1] = eax # rec = rec + sopfr_aux(n/div)
    else:
        eax = stack[display[1]+1] # div
        stack += [eax]           # stash left argument
        eax = 1
        ebx = eax                # right argument
        eax = stack.pop()        # recall left arg
        eax = eax + ebx          # div + 1
        stack[display[1]+1] = eax # div = div + 1
        eax = stack[display[2]+0] # push n
        stack += [eax]           # sopfr_aux(n) call
        sopfr_aux()              #
        del stack[-1:]           # pop n
        stack[display[2]+1] = eax

    eax = stack[display[2]+1]
    display[2] = stack.pop()     # restore frame pointer
    del stack[-1:]              # remove rec from stack

```

```

global display, stack, eax, ebx
stack += [None]*1             # allocate memory for one local variable div
stack += [display[1]]         # save old frame pointer
display[1] = len(stack)-3     # frame pointer for sopfr()
eax = 2
stack[display[1]+1] = eax     # div = 2
eax = stack[display[1]+0]     # push n
stack += [eax]                # sopfr_aux(n)
sopfr_aux()                   #
del stack[-1:]                # pop n
display[1] = stack.pop()      # restore frame pointer
del stack[-1:]                # remove div from stack

```

```

global display, stack, eax, ebx
stack += [None]*0            # no local variables
stack += [display[0]]        # save old frame pointer
display[0] = len(stack)-1    # frame pointer for main()
eax = 42                      # push 42
stack += [eax]               # sopfr(42)
sopfr()                      #
del stack[-1:]               # pop 42
print(eax)
display[0] = stack.pop()     # restore frame pointer

```

```

display = [ None ]*3 # uninitialized frame pointers
stack = []           # empty stack
eax, ebx = None, None # registers

```

```
eax,ebx = None,None # registers
main()
```

► Он же текстом

Подводим итог

Повторюсь, что важно в этом коде, так это повторяемость одних и тех же паттернов тела функции, вызова функции и оценки значения выражений, они в принципе не требуют работы головного мозга, всё пишется исключительно при помощи спинного, что отлично подходит для автоматизации процесса.

[Вот здесь](#) можно посмотреть на все изменения в репозитории с предыдущего релиза, их совсем немного. Версия компилятора для тестирования - [v0.0.4](#).

В итоге мы (почти) не пользуемся ничем от питона, что не умеет ассемблер напрямую. Практически единственное, что нужно для генерации ассемблерного кода - это поменять трафарет, через который мы пишем код. В следующий как раз об этом и поговорим!

Если эта публикация вас вдохновила и вы хотите поддержать автора — не стесняйтесь нажать на кнопку