Компилятор за выходные: таблицы символов

haqreu

Компилятор за выходные: таблицы символов

Средний

9 мин

8.3K

Продолжаем наш вечерний концерт по заявкам радиослушателей. Тема сегодняшнего разговора - таблицы символов. Напоминаю, что в прошлые разы мы поговорили о синтаксических деревьях и способе их построения из исходника мной придуманного языка wend (сокращение от week-end). Вот краткое оглавление серии статей:

Синтаксические деревья и наивный транслятор в питон

Лексер/парсер

2'. Про́клятый огонь, или магия препроцессора C

Таблицы символов: области видимости переменных и проверка типов (эта статья)

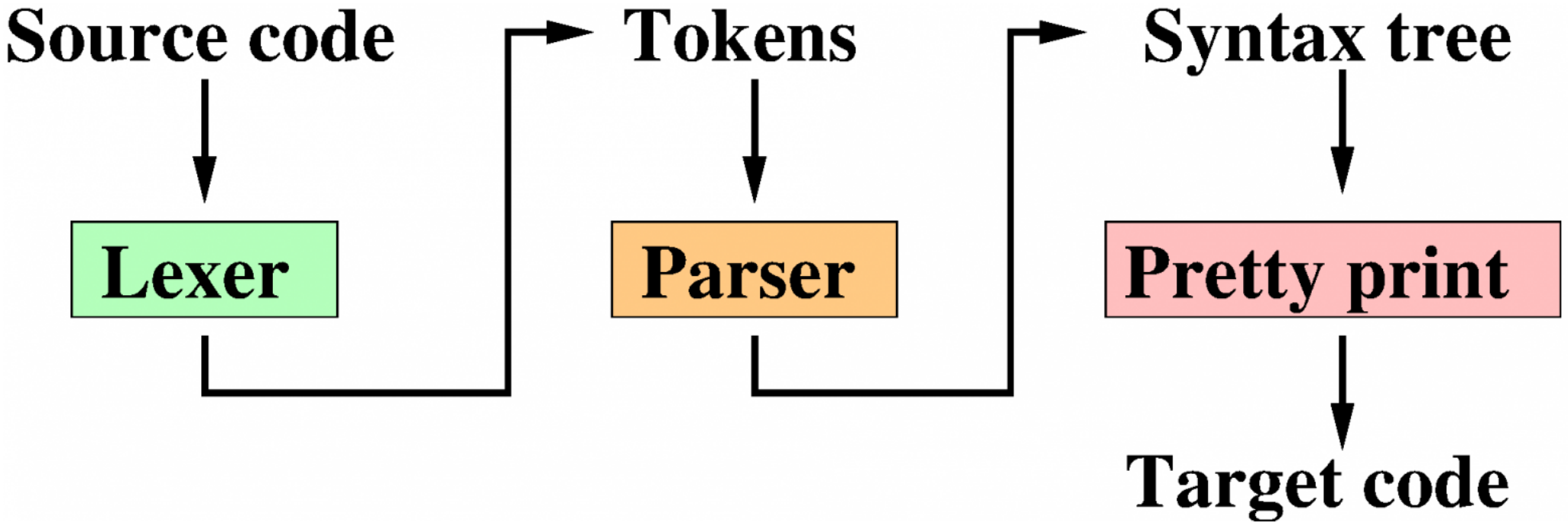
Стек и транслятор в питон без использования питоновских переменных

Транслятор в ассемблер

Рейтрейсинг :)

Вообще целевым языком является ассемблер, но пока что в качестве промежуточного результата я генерирую код на питоне. На

данный момент вся генерация кода - это просто pretty print надстройка над синтаксическим деревом. Напоминаю, что на данный момент процесс "компиляции" выглядит следующим образом:



Лексер превращает поток символов исходного кода в поток лексем, парсер их разбирает, строя синтаксическое дерево. Ну а затем простым обходом дерева в глубину я выдаю код на целевом языке. В простых случаях это неплохо работает, но в конце прошлой статьи я специально оставил пару случаев, в которых компилятор ломается. Давайте вспомним один из них, слева исходник на wend, справа неверная трансляция в питон:

```
fun main() {
    var x:int;

    fun f(y:int) {
        fun g(y:int) {
            x = y + 1;
        }

      x = y;
      g(x + 1);
    }
```

```
def main():
.   x = None

.   def f(y):
.   .   def g(y):
.   .   .   x = (y) + (1)


.   .   x = y
.   .   g((x) + (1))
```

Я ожидаю, что код выведет на экран 3, в то время как питон мне показывает 0. Почему? Давайте разбираться. Для начала отложим wend в сторону и просто поговорим о питоне.

Области видимости переменных в питоне

Никакой Америки я не открою, но я с удивлением для себя обнаружил, что изрядное количество людей не знает, как работает связывание переменных в этом языке. Давайте рассмотрим простейший пример: у меня есть функция `foo()`, которая выводит на экран значение переменной `bar`, которая определена вне функции:

```
ssloy@home:~$ python3 <<<'
def foo():
    print(bar)
bar = 0
foo()
print(bar)
'
0
0
```

Как и раньше, я разом привожу и код, и результат его выполнения. Вполне ожидаемо, что язык с неявным заданием переменных найдёт глобальную переменную `bar` и выведет на экран два нуля. А что будет, если я попытаюсь не только вывести значение `bar`, но

ещё и записать в неё единицу?

```
ssloy@home:~$ python3 <<<'
def foo():
    print(bar)
    bar = 1
bar = 0
foo()
print(bar)
'
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
  File "<stdin>", line 3, in foo
UnboundLocalError: cannot access local variable 'bar' where it is not associated with a value
```

А случилась у нас ошибка компиляции, причём именно компиляции, а не вывалилось во время исполнения. Если же внутри функции переставить операции присваивания и вывода на экран, то всё прекрасно запускается, причём явно значение глобальной переменной `bar` не меняется:

```
ssloy@home:~$ python3 <<<'
def foo():
    bar = 1
    print(bar)
bar = 0
foo()
print(bar)
'
1
```

```
0
```

Это совершенно не магия, это абсолютно нормально, и одновременно крайне контринтуитивно для новичков, пришедших в питон из языков с более строгим объявлением переменных. Питон, как и большинство других языков, разделяет переменные на локальные и на глобальные, причём **на чтение контекст глобальный, а на запись локальный**. Вот и получается, что если мы сначала попытаемся сделать `print(bar)`, а потом присвоить `bar = 1`, то питон знает, что `bar` - локальная переменная, но ещё не была инициализирована, и ломается. Можно использовать слово `global`, чтобы явно указать, что `bar` должна быть глобальной переменной несмотря на то, что в неё идёт запись:

```
ssloy@home:~$ python3 <<<'
def foo():
    global bar
    bar = 1
    print(bar)
bar = 0
foo()
print(bar)
'
1
1
```

Моя практика показывает, что обычно люди и ограничиваются двумя ключевыми словами `global`/`local`, забывая ещё про одно крайне любопытное: `nonlocal`. Давайте рассмотрим следующий пример кода:

```
ssloy@home:~$ python3 <<<'
def counter():
    count = -1
    def increment():
```

```
        nonlocal count
        count += 1
        return count
    return increment
counter1 = counter()
counter2 = counter()

for _ in range(3):
    print("outer counter:", counter1())
    for _ in range(2):
        print("  inner counter:", counter2())
'

outer counter: 0
  inner counter: 0
  inner counter: 1
outer counter: 1
  inner counter: 2
  inner counter: 3
outer counter: 2
  inner counter: 4
  inner counter: 5
```

Здесь есть функция `counter()` со вложенной функцией `increment()`, при этом `increment()` ссылается на переменную `count`, которая не является ни локальной для `increment()`, ни глобальной. Переменная `count` является локальной для функции `counter()`, и `increment()` ссылается на локальную переменную окружающего контекста.

Я создал два счётчика `counter1` и `counter2`, они могут считать независимо друг от друга, поскольку ссылаются на два разных экземпляра переменной `count`. Эта магия называется замыканием, и это исключительно мощный инструмент, который использовать

нужно, но осторожно, чтобы не ломать мозг тем, кто будет поддерживать ваш код :)

Возвращаясь к нашим баранам, вот так должен выглядеть корректный перевод с wend на питон, обратите внимание на появление двух строк с ключевым словом `nonlocal`:

```
fun main() {
    var x:int;

    fun f(y:int) {
        fun g(y:int) {
            x = y + 1;
        }

        x = y;
        g(x + 1);
    }

    x = 0;
    f(x + 1);
    println x;
}
```

"scope.wend"

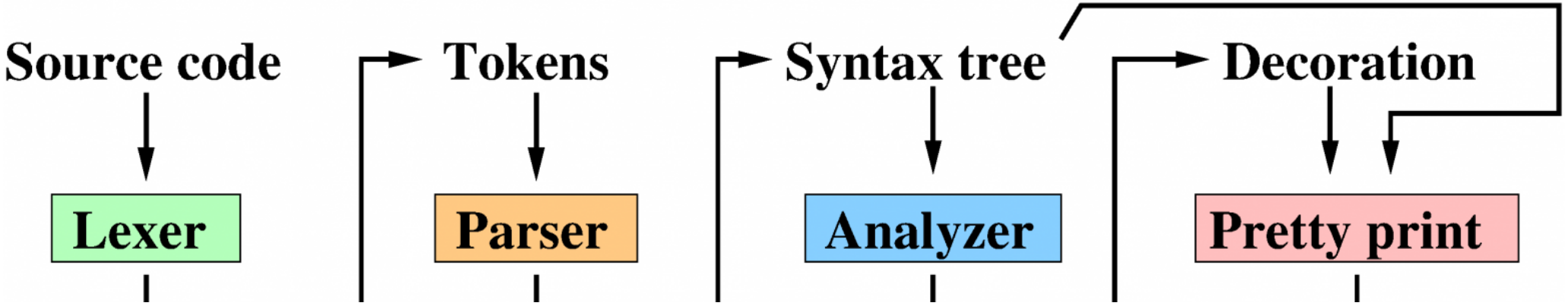
```
def main():
    pass # no nonlocal variables
    x = None
    def f(y):
        nonlocal x
        def g(y):
            nonlocal x
            x = (y) + (1)
        x = y
        g((x) + (1))


    x = 0
    f((x) + (1))
    print(x, end='\n')
main()
```

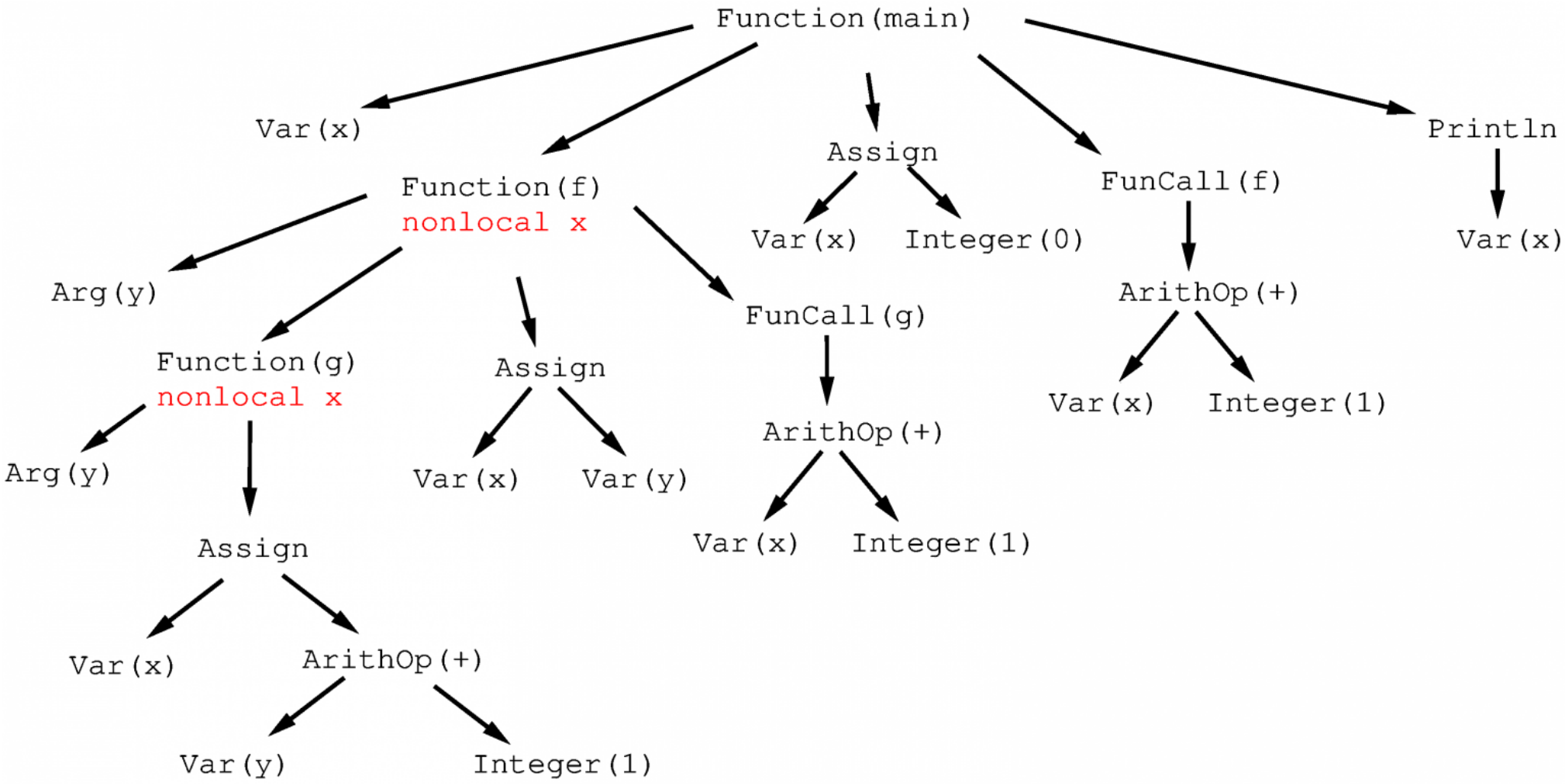
"scope.py"

Ёлочные игрушки

Теперь давайте разбираться, как научить компилятор добавлять подобную информацию. Нам нужно изменить процесс компиляции, добавив один этап семантического анализа, который будет вызван непосредственно после построения синтаксического дерева:



Парсер нам выращивает новогоднюю ёлку, а семантический анализатор развешивает на ней ёлочные игрушки (кстати, это не шутка, компиляторщики реально пользуются таким жаргоном). Ну а дальше генерация кода на целевом языке идёт из украшенной ёлки. Вот так выглядит синтаксическое дерево для нашего примера после прохода семантического анализатора:

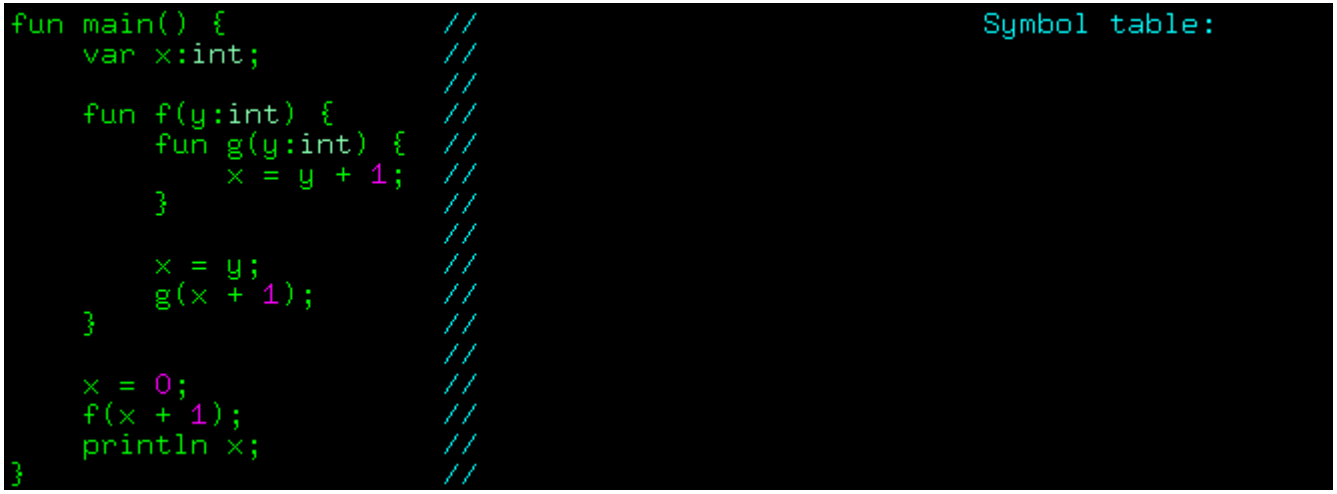


Обратите внимание на информацию, помеченную красным, она и была добавлена анализатором. Самое удобное место для её хранения - это словарь `deco`, который я предусмотрел для каждого узла дерева. Пока что я там хранил всякое типа номера строки в исходном файле для сигнализирования об ошибках, а теперь оно пригодится для семантического анализа.

Надеюсь, что в целом понятно, какую информацию нам нужно добавить дереву. Но как именно это сделать? И тут приходят на помощь таблицы символов.

Таблицы символов

Давайте я отрисую анимацию работы примитивного семантического анализатора:



Мне нужна структура данных, в которую я буду постепенно добавлять (и удалять!) информацию об областях видимости переменных. Делать я это буду, обходя синтаксическое дерево в глубину. Для компактности отображения в моей анимации нарисовано не само синтаксическое дерево, а исходный код, но надо понимать, что он к этому моменту давно выкинут, и работаю я с деревом. Просто проход по строчкам исходного кода сильно легче нарисовать, а он в точности соответствует обходу синтаксического дерева в глубину.

В моём языке области видимости в точности соответствуют функциям, так что, начав с корня дерева, я открываю новую область видимости переменных: `push_scope(main)`. Затем я добавляю в мою таблицу все локальные переменные: `add_var(x)`. На моей анимации я отрисовал тип переменной, поскольку он всё равно маячит в коде, но на данный момент он мне не нужен, я на него буду

смотреть когда-нибудь потом, когда займусь проверкой типов.

Следующий узел в моём синтаксическом дереве - вложенная функция `f`. Открываем новую область видимости, создавая вложенную таблицу: `push_scope(f)`, и добавляем в неё все локальные переменные, тут только один аргумент `add_var(y)`. Аналогично происходит и с функцией `g`: `push_scope(g)`, `add_var(y)`.

И вот тут самое интересное: мы встречаемся с узлом `Assign`, который соответствует строчке `x = y + 1`. В нашем синтаксическом дереве о переменной `x` мы знаем только её идентификатор, просто строку, ничего больше. Давайте узнаем о ней побольше: `find_var(x)`. Мы знаем, что находимся на третьем уровне вложенности, поэтому давайте посмотрим, есть ли в текущей области видимости запись о `x`.? Нет, нету. На втором уровне? Тоже нет. На третьем? Есть, нашлась! Таким образом, мы можем сказать второму и третьему блоку видимости, что в них используется нелокальная переменная `x`.

А заодно (задел на будущее) мы нашли тип переменной, и сейчас можно проверить, совпадает ли тип переменной в инструкции присваивания: мы знаем, что справа стоит `ArithOp`, он обязан быть целочисленным. Если `x` имеет другой тип, самое время свалиться с ошибкой.

Мы разобрались с функциональностью таблицы символов, давайте перейдём к имплементации. Я сделал крайне примитивно:

```
class SymbolTable():
    def __init__(self):
        self.variables = [{}]     # stack of variable symbol tables
        self.ret_stack = [ None ] # stack of enclosing function symbols, useful for return statements
    def add_var(self, name, deco):
        if name in self.variables[-1]:
            raise Exception('Double declaration of the variable %s' % name)
        self.variables[-1][name] = deco

    def push_scope(self, deco):
        self.variables.append({})
        self.ret_stack.append(deco)
```

```
    def pop_scope(self):
        self.variables.pop()
        self.ret_stack.pop()

    def find_var(self, name):
        for i in reversed(range(len(self.variables))):
            if name in self.variables[i]:
                return self.variables[i][name]
        raise Exception('No declaration for the variable %s' % name)
```

Я храню вложенные области видимости как список словарей, ключами которых являются идентификаторы, а значениями - украшение `deco` соответствующего узла синтаксического дерева (в `deco` хранится тип переменной). При входе в область видимости я добавляю словарь (`push_scope`), при выходе удаляю (`pop_scope`). Ну и параллельно я храню такой же список украшений из узлов-функций, что позволит синтаксическому анализатору добавить в дерево информацию о нелокальных переменных.

Вот так выглядит код семантического анализатора, это просто примитивный обход дерева в глубину, который делает запросы к таблице символов каждый раз, как встречает какую-нибудь переменную:

▸ Hidden text

Рабочий коммит доступен здесь, теперь scope.wend компилируется корректно!

Проверка типов и перегрузка функций

Мы уже встретились с рудиментарной проверкой типов переменных, но для полной картины мира нам нужно ещё добавить в таблицу символов функции, ведь когда мы встречаемся с вызовом `f(x+1)`, то мы не знаем ничего про тип, возвращаемый `f`, потому что узел дерева`FunCall` хранит только идентификатор `f`, ничего больше. Всё крайне тривиально: параллельно списку словарей символов переменных давайте заведём список словарей символов функций, и будем их добавлять перед открытием соответствующей области видимости.

В словаре переменных ключом являлся идентификатор переменной, с функциями самую малость сложнее: я хочу уметь перегружать

функции, поэтому идентификатор не является уникальным ключом. Не страшно, я в качестве ключа буду хранить сигнатуру функции, которая является простым кортежем (идентификатор, список типов аргументов).

Я добавил десяток строчек в модуль symtable.py и накидал исключений при несоотвествии типов в семантический анализатор analyzer.py, смотрите на изменения в коде.

Последним штрихом я добавил перегрузку функций: для этого мне достаточно к имени функции приклеить уникальный суффикс, и вуаля, у нас есть полностью корректно работающий компилятор из wend в python!

```
fun main() {
    fun main(x:int) : int {
        return x;
    }

    fun main(x:int, y:int) : int {
        return x + y;
    }

    fun main(x:bool) : int {
        if x {
            return 0;
        }
        return 2;
    }

    println main(0);
    println main(0, 1);
    println main(false);
}
```

```
def main_uniqstr31():
    pass # no non-local variables
    def main_uniqstr32(x):
        pass # no non-local variables
        return x
    def main_uniqstr33(x, y):
        pass # no non-local variables
        return (x) + (y)
    def main_uniqstr34(x):
        pass # no non-local variables
        if x:
            return 0
        else:
            pass
        return 2

    print(main_uniqstr32(0), end='\n')
    print(main_uniqstr33(0, 1), end='\n')
    print(main_uniqstr34(False), end='\n')
main_uniqstr31()
```

Текущий код брать по тегу v0.0.3.

В следующем выпуске

В этот раз мы починили компилятор, использовав ключевое слово `nonlocal`. Но давайте не будем терять из виду то, что вообще-то целевым языком является не питон, а ассемблер, а он про замыкания не знает уж точно ничего! Поэтому в следующий раз мы поговорим про генерацию кода по-прежнему на питоне, но в принципе без использования переменных внутри функций. У меня будут только четыре глобальных переменных, и помимо них никакой другой использовать нельзя: я буду эмулировать регистры и стек. И вот

тут выдаваемый код по-настоящему потеряет читаемость, так что, снявши голову, по волосам не плачут, можно уже и ассемблер генерировать :)

Слева направо: исходник на wend, трансляция в питон с использованием ключевого слова `nonlocal`, трансляция в питон с использованием стека, структурно очень близкий к ассемблерному коду.

```
fun main() {
    var x:int;

    fun f(y:int) {
        fun g(y:int) {
            x = y + 1;
        }

        x = y;
        g(x + 1);
    }

    x = 0;
    f(x + 1);
    println x;
}
```

```
def main():
    x = None
    def f(y):
        nonlocal x
        def g(y):
            nonlocal x
            x = (y) + (1)
        x = y
        g((x) + (1))


    x = 0
    f((x) + (1))
    print(x, end='\n')
main()
```

```
def main_uniqstr28():
    global eax, ebx, stack, display
    def f_uniqstr29():
        global eax, ebx, stack, display
        def g_uniqstr30():
            global eax, ebx, stack, display
            eax = stack[display[2]+0]
            stack.append(eax)
            eax = 1
            ebx = eax
            eax = stack.pop()
            eax = eax + ebx
            stack[display[0]+0] = eax
        eax = stack[display[1]+0]
        stack[display[0]+0] = eax
        eax = stack[display[0]+0]
        stack.append(eax)
        eax = 1
        ebx = eax
        eax = stack.pop()
        eax = eax + ebx
        stack.append(eax)
        stack.append(display[2])
        display[2] = len(stack)-2
        eax = g_uniqstr30()
        display[2] = stack.pop()
        del stack[-1]
    eax = 0
    stack[display[0]+0] = eax
    eax = stack[display[0]+0]
    stack.append(eax)
    eax = 1
    ebx = eax
    eax = stack.pop()
    eax = eax + ebx
    stack.append(eax)
    stack.append(display[1])
    display[1] = len(stack)-2
```

```
    eax = f_uniqstr29()
    display[1] = stack.pop()
    del stack[-1]
    eax = stack[display[0]+0]
    print(eax, end='\n')
eax, ebx = None, None
display = [ 65536 ]*3
stack = []
display[0] = len(stack)
stack.append( None )
main_uniqstr28()
```

Развлекуха

Ну и в качестве финальной развлекухи я позволил себе запрограммировать фантазию на тему игры arcanoid:

Если кому интересно, то вот соответствующий код на C, всего 65 строчек. Любые мысли по улучшению приветствуются, а также кидайте идеи интересного короткого кода!

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