

Компилятор за выходные: лексер и парсер

haqreu

Компилятор за выходные: лексер и парсер

Средний

12 МИН

17K

[illegible]

Продолжаем разговор. На прошлой неделе я пообещал за выходные написать компилятор из простенького мной придуманного языка в ассемблер. В назначенное время уложился, и компилятор даже вроде работает, см. заглавную картинку. Теперь дело за малым, потихоньку причесать и стройно изложить. В прошлый раз я рассказал про синтаксические деревья и показал простейший транслятор в питон (по факту, обычный pretty print дерева). Но если в предыдущей статье я синтаксическое дерево строил вручную, то сегодня всё же будем автоматизировать процесс.

Сегодня я публикую две статьи разом, поскольку по дороге меня довольно круто занесло, и [получился небольшой спин-офф](#). Очень рекомендую к прочтению :)

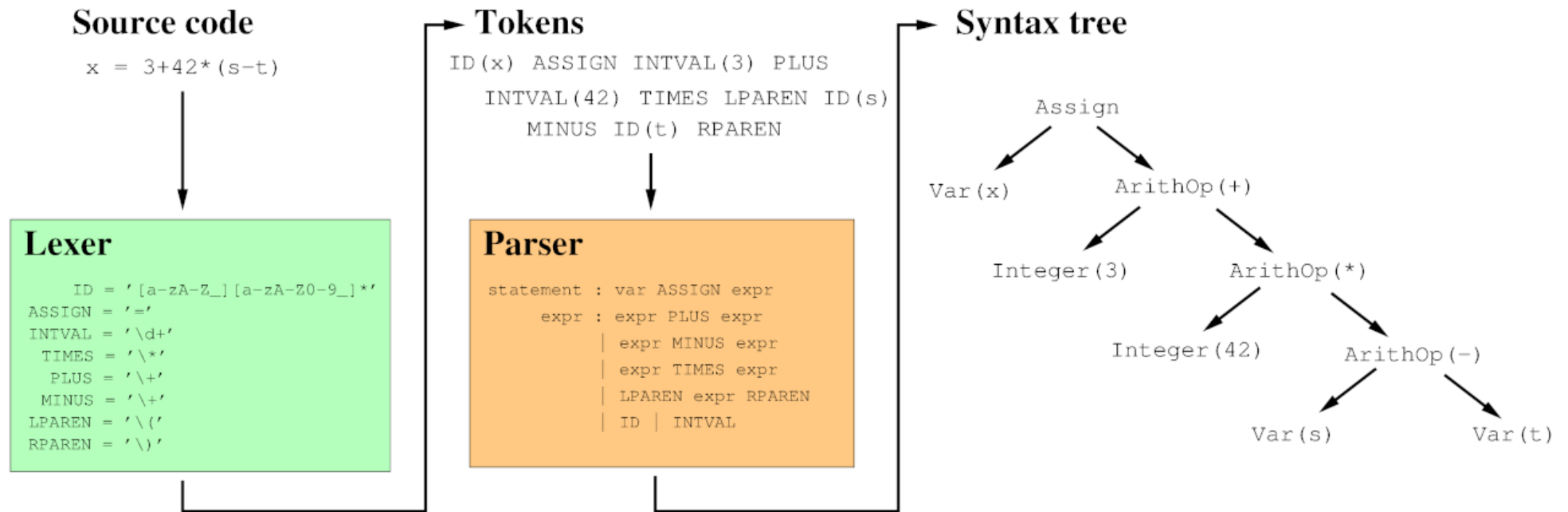
Ну а тема этой статьи - автоматическое построение синтаксического дерева ака лексер и парсер.

1. [Синтаксические деревья и наивный транслятор в питон](#)
2. **Лексер/парсер (эта статья)**
 - 2'. [Проклятый огонь, или магия препроцессора C](#)
3. [Таблицы символов: области видимости переменных и проверка типов](#)
4. [Стек и транслятор в питон без использования питоновских переменных](#)
5. Транслятор в ассемблер
6. Рейтрейсинг :)

Поехали. Компилятор переводит код с исходного языка (обычно языка программирования высокого уровня) на целевой язык (обычно язык низкого уровня), чтобы создать исполняемую программу. Типичный компилятор достигает этого, выполняя ряд шагов, из которых нас сегодня интересуют два первых:

1. **Лексический анализ:** принимает поток символов (исходный код) и выдает поток лексем.
2. **Синтаксический анализ:** анализирует синтаксическую структуру данного потока лексем, чтобы проверить, был ли исходный код синтаксически корректен в соответствии с правилами исходного языка, и выдает синтаксическое дерево, соответствующее исходному коду.

Схематично это выглядит как-то так, я нарисовал вполне конкретный пример из моего компилятора, вы уже встречались с этими именами классов:



В более сложных языках вполне могут быть циклы между парсером и лексером, но я на этом останавливаться не буду. Ещё раз, наша задача - из исходного кода построить синтаксическое дерево, и для этого я буду пользоваться внешним инструментом, которому задам набор лексических правил а также грамматику моего языка (начинка зелёной и оранжевой коробочки на рисунке).

Давайте рассмотрим обе коробочки подробнее.

Лексический анализ

Лексический анализ - это процесс преобразования потока символов в последовательность лексем (токенов). Токен представляет собой строку с заданным значением, которая описывает ряд связанных лексем. Лексема - это последовательность символов, которая образует лексическую единицу в грамматике языка. Интуитивно ее можно рассматривать как "слово" в человеческом языке. Чтобы понять эти определения, давайте рассмотрим таблицу ниже:

Лексемы	Тип лексемы	Семантическое значение
if, while, fun, var, return, print	IF, WHILE, FUN, VAR, RETURN, PRINT	Зарезервированные слова
i tmp for23	ID	Идентификатор
13,27	INTVAL	Целочисленная константа
{,}	LPAREN, BEGIN, END	Символы
%,+/,=	MOD, PLUS, DIVIDE, ASSIGN	операторы

Таким образом, лексер должен принимать поток символов из исходного кода и выдавать последовательность лексем. Например, рассмотрим следующий исходный код:

```
a = 2;
```

Наш лексер должен выдать следующую последовательность лексем:

```
ID(a) ASSIGN INTVAL(2) SEMICOLON
```

Обратите внимание, что внутри лексем спрятаны данные о них: тип лексемы и непосредственно содержание, мы же не должны потерять ни а, ни 2. Мы просто разбили входной текст на слова, при этом присвоив каждому слову какой-то тип (например, в человеческом языке это может быть существительное, глагол).

Как же наш лексер может это сделать? Другими словами, как мы можем описать и распознать лексемы в языке программирования? Дайте-ка я процитирую коллегу Hadrien Titeux:



Parsing: what computer scientists solved 40 years ago, but you still can't do it easily on your own.

Лексемы описываются с помощью регулярных выражений и распознаются конечными автоматами. Например, следующее регулярное выражение может использоваться для определения валидного идентификатора: $[a-zA-Z_][a-zA-Z0-9_]*$. То есть, можно использовать букво-цифры и подчёркивание. При этом первый символ не может быть цифрой, всё стандартно. И тут мы сразу же врезаемся в

Неоднозначности

Что делать, если мы столкнемся со следующим потоком символов?

```
inta=0;
```

Это можно интерпретировать двояко, порождая разные потоки лексем:

```
int | a | = | 0 | ;    => TYPE(int) ID(a) ASSIGN INTVAL(0) SEMICOLON
inta | = | 0 | ;      => ID(inta) ASSIGN INTVAL(0) SEMICOLON
```

Как наш лексер может решить, какие правила использовать (ID или TYPE)? Интуитивно понятное решение - использовать то правило, которое соответствует большему количеству символов в исходном коде. В нашем случае будет использовано правило ID, потому что оно соответствует 4 символам (inta), в то время как правило int соответствует только 3 символам (int). Однако есть и другой случай неоднозначности, который может обмануть наш лексер:

```
int=0;
```

Какой поток лексем соответствует этому исходному коду?

```
TYPE(int) ASSIGN INTVAL(0) SEMICOLON
```

или

```
ID(int) ASSIGN INTVAL(0) SEMICOLON
```

Обратите внимание, что даже если приведенный выше код синтаксически неверен, в данный момент мы занимаемся лексическим анализом, а не синтаксическим, поэтому даже не пытаемся проверять на этом этапе на синтаксические ошибки. Чтобы решить эту проблему, мы будем использовать приоритет правил, то есть указывать, что определенное правило должно быть применено, если два или более регулярных выражения совпадают с одинаковым количеством символов из строки.

SLY: Lexer

Для простоты (см. предыдущую картинку) я буду опираться на уже существующие инструменты для создания синтаксического дерева, а именно, на питоновскую библиотеку SLY. Она сама по себе всего на пару тысяч строк, так что можно было бы для разбора wend сделать всё с нуля, но, пожалуй, ни к чему это. Я свои лекции веду больше в область компиляции, а не в область формальных языков, так что позволю себе зависимость.

SLY предоставляет два отдельных класса Lexer и Parser. Класс Lexer используется для разбиения входного текста на набор лексем, заданных набором правил регулярных выражений. Класс Parser используется для распознавания синтаксиса языка, заданного в виде контекстно-свободной грамматики. Эти два класса используются вместе для создания синтаксического анализатора.

Давайте начнём с лексера. Описание лексических правил тривиально, поэтому я приведу их для всего моего языка целиком, а не буду строить постепенно, начиная с калькуляторов выражений:

```
from sly import Lexer

class WendLexer(Lexer):
    tokens = { ID, BOOLVAL, INTVAL, STRING, PRINT, PRINTLN, INT, BOOL, VAR, FUN, IF, ELSE, WHILE, RETURN,
               PLUS, MINUS, TIMES, DIVIDE, MOD, LTEQ, LT, GTEQ, GT, EQ, NOTEQ, AND, OR, NOT,
               LPAREN, RPAREN, BEGIN, END, ASSIGN, SEMICOLON, COLON, COMMA }
    ignore = '\t\r'
    ignore_comment = r'\/\/.*'

    ID      = r'[a-zA-Z][a-zA-Z0-9_]*' # a regex per token (except for the remapped ones)
    INTVAL  = r'\d+'                  # N. B.: the order matters, first match will be taken
    PLUS    = r'\+'
    MINUS    = r'\-'
```

```

TIMES      = r'\*'
DIVIDE     = r'/'
MOD        = r'%'
LTEQ       = r'<='
LT         = r'<'
GTEQ       = r'>='
GT         = r'>'
EQ         = r'=='
NOTEQ      = r'!='
AND        = r'\&\&'
OR         = r'\|\|'
NOT        = r'!'
LPAREN     = r'\('
RPAREN     = r'\)'
BEGIN      = r'\{'
END        = r'\}'
ASSIGN     = r'='
COLON      = r':'
SEMICOLON  = r';'
COMMA      = r','
STRING     = r'"[^"]*"''

ID['true']  = BOOLVAL # token remapping for keywords
ID['false'] = BOOLVAL # this is necessary because keywords match legal identifier pattern
ID['print'] = PRINT
ID['println'] = PRINTLN
ID['int']   = INT
ID['bool']  = BOOL
ID['var']   = VAR
ID['fun']   = FUN
ID['if']    = IF
ID['else']  = ELSE
ID['while'] = WHILE
ID['return'] = RETURN

@(r'\n+')
def ignore_newline(self, t): # line number tracking
    self.lineno += len(t.value)

def error(self, t):
    raise Exception('Line %d: illegal character %r' % (self.lineno, t.value[0]))

```

Для начала (строки 4-6) надо определить все возможные типы лексем (бонусное чтение: [почему не возникает NameError?](#)). Затем (строки 7-8) мы просим SLY игнорировать пробелы (мы же не в питоне, в конце-то концов!), и выкидывать до конца строки всё, что начинается с двух слешей. А затем просто даём список всех регулярных, соответствующих каждой лексеме.

Обратите внимание: порядок этих регулярных важен. В большой семье валенки одни, кто первый встал, того и сапоги. Давайте для примера посмотрим на лексемы NOT и NOTEQ. Нам необходимо правило NOTEQ обрабатывать раньше, нежели правило NOT, иначе последовательность символов `a != 0` разобьётся на лексемы `ID(a) NOT ASSIGN INTVAL(0)`, а не на требуемое `ID(a) NOTEQ INTVAL(0)`.

Второй важный момент - это то, что все зарезервированные слова `wend` являются валидными идентификаторами, и поскольку правило `ID` стоит самым первым, то таковыми и будут определены. Можно было бы завести на них отдельные правила, но SLY предлагает механизм переназначения лексем, который я и использую (строки 36-47).

Собственно, и всё, больше ничего интересного в списке лексических правил нет. Сам по себе лексер берёт эти правила, и создаёт конечный автомат, который и обрабатывает весь поток символов.

SLY: Parser

Задача синтаксического анализатора - получить поток лексем и проверить, что лексемы образуют допустимое выражение в соответствии со спецификацией языка исходного кода. Обычно это делается с помощью контекстно-свободной грамматики, которая рекурсивно определяет компоненты, из которых может состоять выражение, и порядок их появления.

Таким образом, синтаксический анализатор должен определить, принимает ли грамматика языка заданный поток лексем, затем построить синтаксическое дерево и передать его остальным частям компилятора. Опять же, я не буду останавливаться на классификациях грамматик и алгоритмах обработки потока лексем, оставим для тех, кто хочет поговорить о теории формальных языков. Единственное, что нам нужно знать, что мы пользуемся [самым простым парсером](#) и для принятия решения о том, что мы должны делать с лексемой, мы не имеем права смотреть дальше, чем на один токен вперёд.

Грамматика моего языка крайне примитивная, и написание соответствующих правил для парсера не должно составить никакого труда. Давайте посмотрим на структуру кода, которого хватит для разбора выражения $x = 3 + 42 * (s - t)$ из картинки в начале статьи (та, что с оранжевой коробочкой):

```
from sly import Parser
from lexer import WendLexer
from syntree import *

class WendParser(Parser):
    tokens = WendLexer.tokens
    precedence = (
        ('left', PLUS, MINUS),
        ('left', TIMES),
    )

    @_('ID ASSIGN expr SEMICOLON')
    def statement(self, p):
        return Assign(p[0], p.expr, {'lineno':p.lineno})

    @_('expr PLUS expr',
        'expr MINUS expr',
        'expr TIMES expr')
    def expr(self, p):
        return ArithOp(p[1], p.expr0, p.expr1, {'lineno':p.lineno})

    @_('LPAREN expr RPAREN')
    def expr(self, p):
        return p.expr

    @_('ID')
    def expr(self, p):
        return Var(p[0], {'lineno':p.lineno})

    @_('INTVAL')
    def expr(self, p):
        return Integer(int(p.INTVAL), {'lineno':p.lineno})

    def error(self, token):
        if not token:
            raise Exception('Syntax error: unexpected EOF')
        raise Exception(f'Syntax error at line {token.lineno}, token={token.type}')
```

Мы наследуемся от класса Parser, и определяем грамматические правила. У нас есть два нетерминальных символа: statement, соответствующий инструкции, и expr, соответствующий выражению. Терминальных символов всего два: INTVAL и ID.

Если парсер увидит лексему INTVAL(строки 30-32), то он ничего дальше не смотрит, и создаёт объект (узел синтаксического дерева) класса Integer. Если он видит лексему ID(строки 26-28), то создаёт узел синтаксического дерева класса Var. Обратите внимание, что в обоих случаях вызывается функция expr, то есть, и целочисленная константа, и переменная у нас являются тривиальными выражениями. Аналогично обрабатываются потоки лексем типа ID(s) MINUS ID(t) - парсер выдаёт узел синтаксического дерева типа ArithOp с двумя потомками ID, которые были найдены рекурсией в парсере. Со скобками и другими арифметическими операциями, думаю, разберётесь :)

Единственное, на что следует обратить внимание, так это на приоритет операций, который я задал в строках 7-10.

Инструкция пока что у нас только одна, так что она может быть построена исключительно тогда, когда поток входных лексем имеет вид `ID ASSIGN expr SEMICOLON`, где `expr` - это последовательность лексем, которая соответствует арифметическому выражению. Надлежащее грамматическое правило и дано в строках 12-14, которые при соответствии такому потоку лексем создаёт узел синтаксического дерева класса `Assign`.

Если вам понятен этот код, то вот полный набор грамматических правил для `wend`, больше там нет никаких тонкостей, разве что превращение унарной операции `-x` в бинарную операцию `0-x`:

► Hidden text

Что дальше?

Соответствующий код [доступен по тегу v0.0.2](#) в репозитории, я больше не буду трогать ни парсер, ни лексер: у нас есть прекрасный инструмент, позволяющий автоматически создавать синтаксические деревья из файлов исходников на `wend`. Протестировать можно, просто запустив `make test`, и увидеть, что базовые тесты проходят, а вот на двух тестах компилятор ломается:

```
ssloy@periwinkle: ~/Downloads/tinycompiler-0.0.2
tests/test_syntree.py::test_syntree PASSED [ 76%]
tests/test_transpy.py::test_transpy[helloworld] PASSED [ 80%]
tests/test_transpy.py::test_transpy[sqrt] PASSED [ 84%]
tests/test_transpy.py::test_transpy[fixed-point] PASSED [ 88%]
tests/test_transpy.py::test_transpy[scope] FAILED [ 92%]
tests/test_transpy.py::test_transpy[overload] FAILED [ 96%]
tests/test_transpy.py::test_transpy[mutual-recursion] PASSED [100%]
```

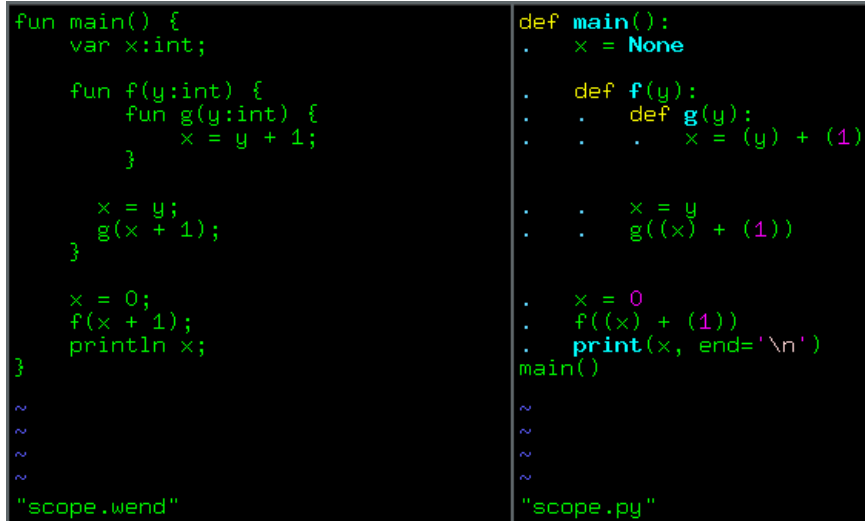
Я напомним, что пока что компилятор - это просто pretty print надстройка над лексером и парсером, пока что я выдаю питоновский код на выход. Давайте поближе посмотрим на проваленные тесты, я даю бок-о-бок исходник `wend` и скомпилированный код на питоне:

<pre>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); } "overload.wend"</pre>	<pre>def main(): . def main(x): . . return x . def main(x, y): . . return (x) + (y) . def main(x): . . if x: . . . return 0 . . else: . . . pass . . return 2 . print(main(0), end='\n') . print(main(0, 1), end='\n') . print(main(False), end='\n') main() "overload.py"</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Питон очень мощный язык, он умеет сильно больше `wend`, и мой pretty print просто тупо повторяет структуру исходника. Единственное, чего не умеет питон - это из коробки

перегружать функции, так что вполне очевидно, что ему не нравятся четыре разные функции `main`, он даже компилироваться отказывается (я говорю про компиляцию питоновским интерпретатором выхода моего компилятора).

А вот тут несколько интереснее:



```
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"
```

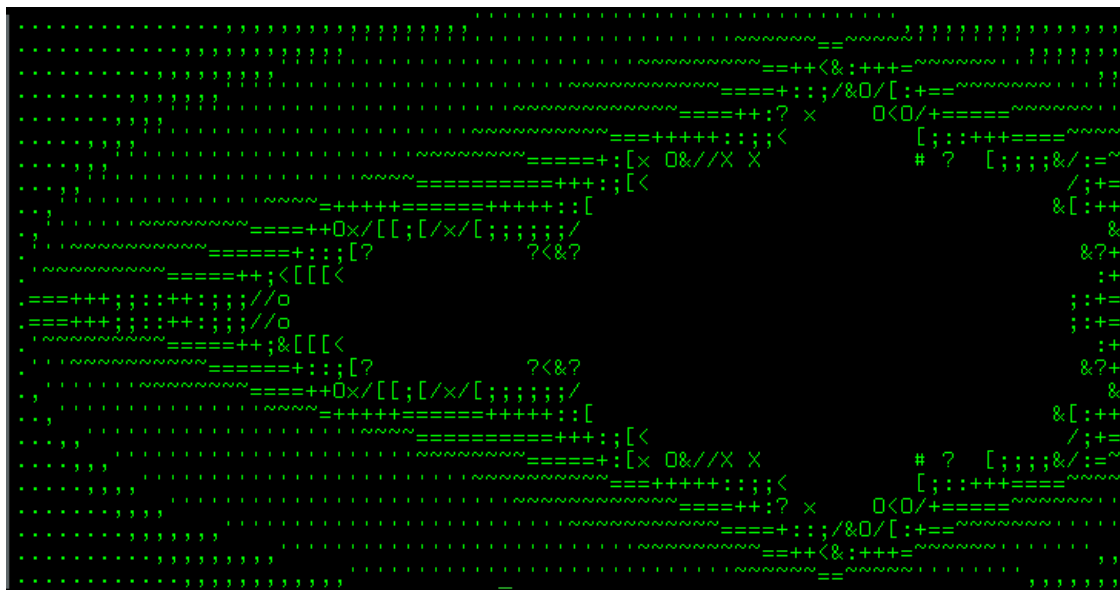
```
def main():  
  x = None  
  
  def f(y):  
    def g(y):  
      x = (y) + (1)  
  
      x = y  
      g((x) + (1))  
  
  x = 0  
  f((x) + (1))  
  print(x, end='\n')  
main()  
~  
~  
~  
~  
"scope.py"
```

Этот питоновский код прекрасно компилируется, но в итоге на экран выводится 0, в то время как ожидаемый вывод моего кода 3. И вот это - тема следующего разговора, а именно, области видимости переменных и таблицы символов. То есть, мы перейдём от лексического и синтаксического анализа к семантическому.

Кстати, множество Мандельброта из заглавной картинки прекрасно компилируется и отображается даже таким примитивным компилятором, вот код:

► Hidden text

А вот ещё раз результат его работы:



Мы начинаем встречать код на wend, для которого было бы очень уж громоздко строить синтаксические деревья вручную, так что парсер - это хорошо!

Послесловие

После лекции о лексере и парсере почитайте мою [сказку о программировании на лексере без использования парсера](#). Она хорошо иллюстрирует происходящее, и при этом заслуженно отправлено в хаб "ненормальное программирование".

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