Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

Отчет по лабораторной работе №2

Лексический анализ

Выполнил:

студент гр. 953504 Басенко К. А.

Проверил:

Шиманский В.В.

Минск 2022

Содержание

Постановка задачи

В данной работе ставится задача - освоить работу с существующими лексическими анализаторами. Разработать собственный лексический анализатор подмножества языка программирования, для чего определить лексические правила и выполнить перевод потока символов в поток лексем (токенов). Основной целью работы является реализация лексического анализатора. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называемые лексемами.

Анализируемый код программы на языке C#:

```
1
2   _ = "string string";
3
4   _ =          5 + 5;
5   _ = 1 + 2;
6   _ = 0 -                3;
7   _ = 9 / 5;
8   _ = 4 * 3           ;
9
10  _ = (1 +2) * 3;
11  _ = (9 - 4)/7;
12
13  const int _1 =1; const int _2= 2;const int _3=3;
14
15  var a = 3d;
16  double b = a + 5;
17  var c=(1.5435e6+b)*0.543f;
```

```
18
19   (_, _) = (2f, 2d);
20   (_, _) = (2m, 2u);
21   (_, _) = (2l, 2L);
22   (_, _) = (2ul, 2UL);
23
24   a += c - b;
25   b -= (a + c) * 0.11e2D;
26   c *= a * b;
27
28   _ = a < b;
29   _ = b >= c;
```

```
30
31  int func(int a, int b, int c)
32  {
33      for (int i = 0; i < b; i++)
34      {
35          a--;
36      }
37
38      while (c > 0)
39      {
40          a *= a;
41          c--;
42      }
43
44      return a;
45  }
46
47  _ = func(4, 5, 2);
```

```
48
49  // Лексические ошибки:
50  _ = d + 5;
51  _ = "str;
52  _ = * 8;
53  _ = a b;
54  _ = 1a;
```

Теория

Лексический анализатор — это программа или часть программы, выполняющая лексический анализ. Лексический анализатор обычно работает в две стадии:

a) сканирование
b) оценка

На первой стадии, сканировании, лексический анализатор обычно реализуется в виде конечного автомата, определяемого регулярными выражениями. В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах. Например, токен «целое число» может содержать любую последовательность десятичных цифр. Во многих случаях первый непробельный символ может использоваться для определения типа следующего токена, после чего входные символы обрабатываются один за другим пока не встретится символ, не входящий во множество допустимых символов для данного токена. В некоторых языках правила разбора лексем несколько более сложные и требуют возвратов назад по читаемой последовательности.

Полученный таким образом токен содержит необработанный исходный текст (строку). Для того чтобы получить токен со значением, соответствующим типу (напр. целое или дробное число), выполняется оценка этой строки — проход по символам и вычисление значения.

Токен с типом и соответственно подготовленным значением передаётся на вход синтаксического анализатора.

В информатике лексический анализ — процесс аналитического разбора входной последовательности символов на распознанные группы — лексемы, с целью получения на выходе идентифицированных последовательностей, называемых «токенами» (подобно группировке букв в словах). В простых случаях понятия «лексема» и «токен» идентичны, но более сложные токенизаторы дополнительно классифицируют лексемы по различным типам («идентификатор, оператор», «часть речи» и т.п.). Лексический анализ используется в компиляторах и интерпритаторах исходного кода языков программирования, и в различных парсерах естественных языков.

Как правило, лексический анализ производится с точки зрения определённого формального языка или набора языков. Язык, а точнее его грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

Традиционно принято организовывать процесс лексического анализа, рассматривая входную последовательность символов как поток символов. При такой организации процесс самостоятельно управляет выборкой отдельных символов из входного потока.

Распознавание лексем в контексте грамматики обычно производится путём их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая, согласно грамматике, не может быть идентифицирована как токен языка, обычно рассматривается как специальный токен-ошибка.

Каждый токен можно представить в виде структуры, содержащей идентификатор токена (или идентификатор класса токена) и, если нужно, последовательность символов лексемы, выделенной из входного потока (строку, число и т. д.).

Цель такой конвертации обычно состоит в том, чтобы подготовить входную последовательность для другой программы, например для грамматического анализатора, и избавить его от определения лексических подробностей в контекстно-свободной грамматике (что привело бы к усложнению грамматики).

Фазы

Фаза - это самостоятельная задача в процессе компиляции. Как правило, несколько фаз объединяются в один проход. Фазы лексического анализатора компилятора (также называемый сканером, лексером) переводит входящие данные в форму, более полезную для остальной части компилятора.

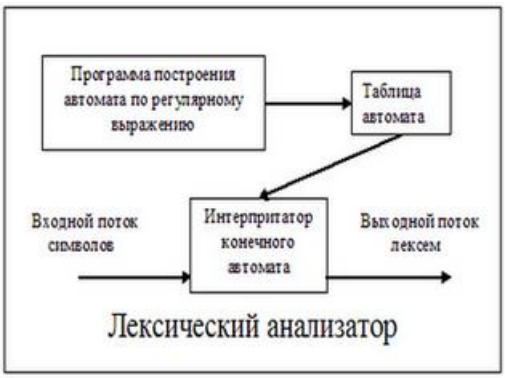
Токены и лексемы

Лексический анализатор просматривает входной поток как совокупность основных элементов языка, называемые токенами. Это означает, что токен является лексической неделимой единицей. В Lisp ключевые слова **while** и **for** являются токенами (мы не можем сказать wh ile), такие символы как >,

>, >= токены, идентификаторы и числа тоже токены и прочее. Исходную строку, содержащую токен, называют лексемой. Обратите внимание, что токен и лексемы похожи, но не одно и то же. Например, токены идентификатор или число могут иметь множество связанных с ними лексем (10, 2.3, 506 - лексемы, число - токен), в тоже время токен совпадает с лексемой состоящей из одного символа. Ситуацию осложняют токены, которые пересекаются с другими (например, какой токен нужно распознать в потоке ">>=": ">", ">>" или ">=" ?). В общем лексический анализатор распознает токен, который соответствует самой длинной лексеме - во многих языках программирование такое поведение указано в спецификации. Таким образом в нашем случае мы получим токен сдвига ">>", а не два токена больше чем.

Выбор множества токенов

Одним из первых проектных решений, которые могут повлиять на структуру всего компилятора, является выбор множества токенов. Вы можете иметь токены для каждаго входного символа или несколько символов могут быть объединены в один токен. Например, символы >, >=, >>, и >>= могут рассматриваться либо как четыре токена, либо как один токен оператор-сравнения. При этом, лексема используется для устранения неоднозначности токена. Первый подход может упростить генерацию кода. Однако, слишком много токенов могут сделать парсер слишком большим и трудным в написании. Нет жестких и быстрых правил для выбора как лучше, но после прочтения книги вы поймете проектные соображения и сможете сделать осознанный выбор. В общем, арифметические операции с одним приоритетом и ассоциативностью могут быть сгруппированы вместе, также ключевые слова для определения типа (вроде int или char) могут быть объединены и т.д.



Анализ текста и разбор кода

Когда программист пытается написать парсер текста, его естественный подход — рекурсивное углубление: найти начало конструкции (например, **{**); найти её конец (например, **}** на том же уровне вложенности); выделить содержимое конструкции, и пропарсить её рекурсивно.

Проблемы с таким подходом — во-первых, избыточная сложность (по одному и тому же фрагменту текста гуляем взад-вперёд); во-вторых, неудобство поддержки (синтаксис языка оказывается рассредоточен по килобайтам и килобайтам ветвистого кода).

Синтаксис языка можно задать декларативно; например, всем знакомы регулярные выражения. Идеально было бы написать стопочку регэкспов для всех конструкций языка, напротив каждого — определение узла, который должен создаться в дереве программы; «универсальный парсер» бы просто подставлял программу в один регэксп за другим, и создавал узлы согласно описанию, один за другим.

Первая из названных проблем решается тем, что поиск всех регэкспов в тексте можно выполнить за один проход, т.е. нет надобности хранить всю программу в памяти целиком — достаточно читать её по одному символу, обрабатывать символ, и тут же забывать.

Вторая — тем, что теперь у нас есть централизованное, формальное описание языка: можем менять регэкспы, вовсе не трогая код; и наоборот, менять код, не рискуя повредить парсер.

Парсер будет читать входную строку символ за символом, и записывать (сдвигать) прочитанные символы в стек. Как только наверху стека соберётся последовательность (символов и переменных), подходящая к правилу грамматики, автомат вытолкнет всю её из стека, и заменит на переменную, стоящую в левой части подошедшего правила

(свёртка). Вся работа автомата заключается в последовательности сдвигов и свёрток.

Интересный момент: автомату на самом деле не важно, какие символы лежат в стеке. Всё равно он не может их сравнить с правилами грамматики, потому что видит только верхний; вместо этого он выбирает, какое правило применить для свёртки, по своему текущему состоянию. Стек ему нужен затем, чтобы знать, в какое состояние перейти после свёртки. Для этого он во время сдвига записывает в стек, вместе с символом, своё текущее состояние; а во время свёртки берёт из стека состояние, записанное под всеми стёртыми символами, и в зависимости от него переходит в следующее состояние.

Результат работы программы

Лексические ошибки:

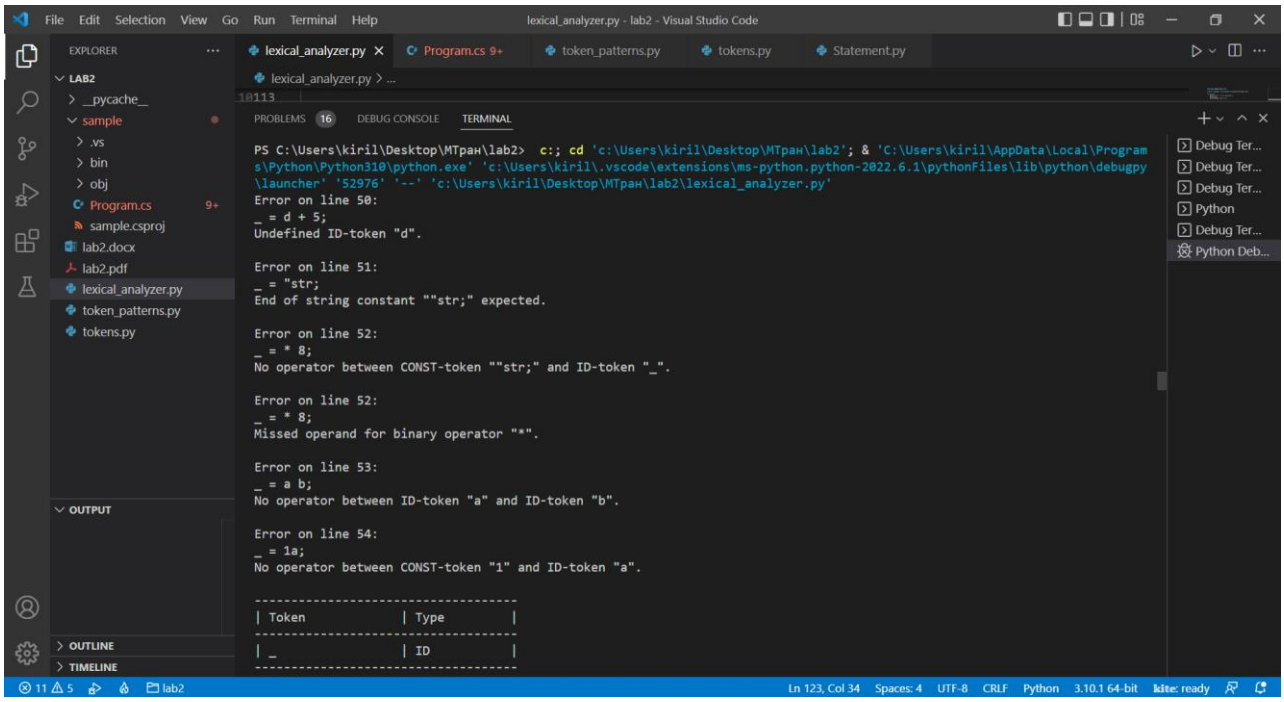


Таблица имен:

Token	Type
_	ID
=	RESERVED
"string string"	CONST
;	RESERVED
_	ID
=	RESERVED
5	CONST
+	RESERVED
5	CONST
;	RESERVED
_	ID
=	RESERVED
1	CONST

+	RESERVED
2	CONST
;	RESERVED
_	ID
=	RESERVED
0	CONST
-	RESERVED
3	CONST
;	RESERVED
_	ID
=	RESERVED
9	CONST
/	RESERVED
5	CONST

;	RESERVED
_	ID
=	RESERVED
4	CONST
*	RESERVED
3	CONST
;	RESERVED
_	ID
=	RESERVED
(	RESERVED
1	CONST
+	RESERVED
2	CONST
)	RESERVED

*	RESERVED
3	CONST
;	RESERVED
_	ID
=	RESERVED
(	RESERVED
9	CONST
-	RESERVED
4	CONST
)	RESERVED
/	RESERVED
7	CONST

;	RESERVED
const	ID
int	ID
_1	ID
=	RESERVED
1	CONST
;	RESERVED
const	ID
int	ID
_2	ID
=	RESERVED
2	CONST
;	RESERVED
const	ID

int	ID
_3	ID
=	RESERVED
3	CONST
;	RESERVED
var	ID
a	ID
=	RESERVED
3d	CONST
;	RESERVED
double	ID
b	ID
=	RESERVED
a	ID
+	RESERVED
5	CONST
;	RESERVED
var	ID
c	ID
=	RESERVED
(	RESERVED
1.5435e6	CONST
+	RESERVED
b	ID
)	RESERVED
*	RESERVED
0.543f	CONST
;	RESERVED
(	RESERVED

_	ID
,	RESERVED
_	ID
)	RESERVED
=	RESERVED
(	RESERVED
2f	CONST
,	RESERVED
2d	CONST
)	RESERVED
;	RESERVED
(	RESERVED
_	ID
,	RESERVED

_	ID
)	RESERVED
=	RESERVED
(	RESERVED
2m	CONST
,	RESERVED
2u	CONST
)	RESERVED
;	RESERVED
(	RESERVED
_	ID
,	RESERVED
_	ID
)	RESERVED
=	RESERVED

(	RESERVED
2l	CONST
,	RESERVED
2L	CONST
)	RESERVED
;	RESERVED
(	RESERVED
_	ID
,	RESERVED
_	ID
)	RESERVED
=	RESERVED
(	RESERVED
2ul	CONST
,	RESERVED
2UL	CONST
)	RESERVED
;	RESERVED
a	ID
+=	RESERVED
c	ID
-	RESERVED
b	ID
;	RESERVED
b	ID
-=	RESERVED
(	RESERVED
a	ID

+	RESERVED
c	ID
)	RESERVED
*	RESERVED
0.11e2D	CONST
;	RESERVED
c	ID
*=	RESERVED
a	ID
*	RESERVED
b	ID
;	RESERVED
_	ID
=	RESERVED
a	ID
<	RESERVED
b	ID
;	RESERVED
_	ID
=	RESERVED
b	ID
>=	RESERVED
c	ID
;	RESERVED
int	ID
func	ID
(	RESERVED
int	ID
a	ID

,	RESERVED
int	ID
b	ID
,	RESERVED
int	ID
c	ID
)	RESERVED
{	RESERVED
for	RESERVED
(	RESERVED
int	ID
i	ID
=	RESERVED
0	CONST
;	RESERVED
i	ID
<	RESERVED
b	ID
;	RESERVED
i	ID
++	RESERVED
)	RESERVED
{	RESERVED
a	ID
--	RESERVED
;	RESERVED
}	RESERVED
while	RESERVED
(	RESERVED

}	RESERVED
while	RESERVED
(	RESERVED
c	ID
>	RESERVED
0	CONST
)	RESERVED
{	RESERVED
a	ID
*=	RESERVED
a	ID
;	RESERVED
c	ID
--	RESERVED
;	RESERVED
}	RESERVED
return	RESERVED
a	ID
;	RESERVED
}	RESERVED
_	ID
=	RESERVED
func	ID
(	RESERVED
4	CONST
,	RESERVED
5	CONST
,	RESERVED

2	CONST
)	RESERVED
;	RESERVED
_	ID
=	RESERVED
d	ID
+	RESERVED
5	CONST
;	RESERVED
_	ID
=	RESERVED
"str;	CONST
_	ID
=	RESERVED
*	RESERVED
8	CONST
;	RESERVED
_	ID
=	RESERVED
a	ID
b	ID
;	RESERVED
_	ID
=	RESERVED
1	CONST
a	ID
;	RESERVED

Вывод

В результате данной работы был разработан лексический анализатор, который не только может составить таблицу имен, но и уведомить об лексических ошибках, выводя их на консоль.