Лекция 7. Методы символьной обработки: рекурсивный спуск и алгоритмы трансляции выражений

К этой области относятся задачи на программирование редактирования текстов, поисковых машин, систем распознавания текстов, на разработку программ-переводчиков, трансляторов языков программирования и других приложений.

Символьная обработка занимает больше времени в компьютере, чем операции с числовой информацией. Входная и выходная информация представляется в текстовом виде и при ее вводе-выводе приходится преобразовывать символьные данные во внутреннее представление и наоборот.

Речевой текст обычно представляет собой последовательность *предложений* – самостоятельных частей, выражающих законченную мысль. Предложения состоят из слов (лексем), составленных из символов некоторого алфавита. *Лексема* (лексическая единица) - это минимальная часть текста, имеющая смысл.

Описание любого разговорного языка (в том числе правила построения входного текста программы) состоит из *грамматики* – описания формы текста, и *семантики*, определяющей смысл этого текста.

Грамматика включает *алфавит* (набор используемых символов), лексику и синтаксис. *Лексика* задает правила записи лексем (слов) из символов алфавита. *Синтаксис* – это правила составления предложений языка из символов и лексем.

Обработка текста включает его *анализ* – разложение на составные части, и действия, зависящие от *семантики* текста – смысла решаемой задачи.

Для описания грамматики текста в программировании широко используются формальные метаязыки: БНФ, МБНФ, регулярные выражения, синтаксические диаграммы и др. Математическим аппаратом символьной обработки является теория формальных языков и грамматик и теория автоматов [66 - 68, 75].

Рассматриваемые методы можно применять для анализа и преобразования не только символьной, но и другой сложно организованной информации.

Синтаксический анализ предложений требует обычно использования более сложных методов, чем обработка слов. В таких случаях программа упрощается, если обработку слов (*лексический анализ*) отделить от обработки предложений.

Тогда программа анализа предложений избавляется от мелочных операций с символами (пропуска пробелов и других “пустых” символов, выделения из текста отдельных слов, имеющих разную длину и т.п.), и работает с лексемами, представленными в виде элементов фиксированной длины.

Одну из проблем техники программирования символьной обработки создает наличие в тексте конструкций переменной длины. Это могут быть, например, числа и другие слова, составленные из разного количества символов, или предложения, состоящие из неизвестного заранее числа лексем.

Каждая часть текста обычно обрабатывается отдельной программой (подпрограммой или фрагментом программы). При наличии в тексте конструкций переменной длины его необходимо читать по одному символу (или по одной лексеме). Чтобы найти конец такой конструкции, программе придется прочитать лишний символ (лексему), относящийся к следующей конструкции, обрабатываемой уже другой программой.

В подобных случаях удобно, чтобы каждая программа обработки части текста вела себя одинаково: начинала работать, когда первый символ ее части прочитан, а заканчивала чтением первого символа следующей части. Тогда эти программы хорошо стыкуются друг с другом. Таким же образом удобно строить тело цикла, обрабатывающего повторяющуюся часть текста.

Если же в таких случаях некоторые программы читают первый символ следующей части, а другие не читают, то могут возникать характерные для символьной обработки неприятные ошибки типа *“плюс-минус один”* с текущим символом, когда программа в какой-то момент читает на один символ больше или меньше требуемого. Эти ошибки трудно не только искать, но и устранять: при исправлении одного места программы нарушается работа в другом.

Рассмотрим простой и достаточно универсальный метод символьной обработки – *рекурсивный спуск* и методы записи и трансляции выражений.

7.1. Метод рекурсивного спуска

Метод *рекурсивного спуска* (метод синтаксических подпрограмм) основан на том, что структура алгоритма часто повторяет структуру читаемых им данных. Повторяющемуся фрагменту данных в алгоритме соответствует цикл, а вариантам представления информации – ветвление.

В методе *рекурсивного спуска* транслятор или другая программа анализа текста представляется в виде набора подпрограмм, каждая из которых читает и обрабатывает в тексте свою конструкцию и вызывает (в том числе рекурсивно) соответствующие подпрограммы для анализа вложенных в нее конструкций.

Формат входных данных описывается в виде грамматики, например, на метаязыке МБНФ. Алгоритмическая структура подпрограммы соответствует правилам грамматики для ее конструкции:

метасимволу повторения “**...**” соответствует цикл,

метасимволу “или” | - условный оператор,

конструкции вида [ ] **…** - цикл с предусловием **while**, т.к. возможно нулевое число повторений, поскольку конструкции вида [ ] - не обязательна.

Пример 7.1. Вычисление выражения. Входной текст представляет собой выражение из целых чисел с операциями +, -, *, / и скобками, заканчивающееся знаком “=”. Требуется составить программу вычисления значения соответствующего выражения.

Пример. Вход: 26+36/2*3-(100+4*5)/30 =
Выход: 76

Решение. Будем решать задачу поэтапно, постепенно расширяя возможности программы. На *первом этапе* будем обрабатывать только выражения с операциями сложения и вычитания без скобок.

Запишем на метаязыке МБНФ грамматику входного текста – выражения. Оно представляет собой алгебраическую сумму слагаемых, являющихся числами. Грамматика примет вид:

выражение ::= слагаемое [{+ | -} слагаемое] **...**
слагаемое ::= число
число ::= цифра **...**
цифра ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Программа будет включать функции (подпрограммы) чтения выражения, слагаемого и числа и вычисления значения соответствующей конструкции.

Здесь имеется конструкция переменной длины: число может состоять из разного количества цифр. Поэтому удобно, чтобы каждая подпрограмма (кроме подпрограммы вычисления выражения, которая вызывается первой) начинала работу, когда предыдущая подпрограмма прочитает первый символ ее конструкции, и заканчивала чтением первого символа следующей конструкции.

Из правил грамматики легко получается набор соответствующих подпрограмм - программа 7.1.

Алгоритм 7.1.

```
/*Вычисление инфиксного выражения без пробелов с целыми     */
/* десятичными числами и операциями +, -.                   */
/* Метод рекурсивного спуска.                                */
#include <stdio.h>
char c;                 /* Текущий символ входного текста    */
int slag ();
int chislo ();

/*    выражение ::= слагаемое [{+ | -} слагаемое]...       */
int vyrazh ()
{  int z, znak;
   c=getchar();
   z = slag();
   while (c=='+' || c=='-')
   {  if (c=='+')  znak=1;  else  znak=-1;
      c=getchar(); z=z+znak*slag();
```

```
   }
   return z;                        /* Значение выражения     */
}
/*    слагаемое ::= число                                      */
int slag ()
{  int z, d;
   z = chislo();
   return z;                        /* Значение слагаемого    */
}

/*    число     ::= цифра...                                   */
int chislo ()
{  int z;
   z=0;
   while (c>='0' && c<='9')
   {  z=10*z+c-'0'; c=getchar(); }
   return z;                        /* Значение числа         */
}
int main ()
{  printf ("%d \n", vyrazh());
   getch();
}
```

На *втором этапе* разрешаем использовать в выражении умножение и деление. Как изменится при этом грамматика входного текста? Очевидно, общий вид выражения сохранится: оно по-прежнему является алгебраической суммой, и подпрограмма vyrazh() останется без изменений.

Изменится только вид слагаемого, которое теперь может содержать операции умножения и деления с числовыми операндами (назовем их “множителями”). В грамматике появятся правила:

слагаемое ::= множитель [{* | /} множитель]**...**
множитель ::= число

Множитель имеет такой же вид, какой на первом этапе имело слагаемое, и в качестве подпрограммы mnozh() для вычисления множителя используем переименованную подпрограмму slag().

Новая подпрограмма slag() очень похожа на подпрограмму vyrazh() – алгоритм 7.2. При делении на ноль программа для простоты не выдает сообщение об ошибке (как должна бы поступать реальная, а не учебная программа), а просто присваивает результату ноль.

Алгоритм 7.2. Версии подпрограмм для реализации умножения и деления.

```
/*    слагаемое ::= множитель [{* | /} множитель]...       */
int slag ()
{  int z, d;
   z = mnozh();
```

```
   while (c=='*' || c=='/')
   {  if (c=='*')
      { c=getchar();  z=z*mnozh(); }
      else
      {  c=getchar(); d=mnozh();
         if (d!=0)  z=z/d;  else  z=0;        /* Ошибка */
      }
   }
   return z;                        /* Значение слагаемого  */
}
/*    множитель ::= число                                    */
int mnozh ()
{  int z, d;
   z = chislo();
   return z;
}                                   /* Значение множителя   */
```

Остался последний шаг – допустить использование скобок. Что теперь произойдет с грамматикой? Изменяется только форма множителя: он может теперь записываться еще и в виде выражения, заключенного в скобки:

множитель ::= число | (выражение)

В подпрограмме вычисления множителя mnozh() появится еще лишь один оператор if. Получим окончательную грамматику:

выражение ::= слагаемое [{+ | -} слагаемое] **...**
слагаемое ::= множитель [{* | /} множитель] **...**
множитель ::= число | (выражение)
число ::= цифра **...**
цифра ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Приоритеты операций отражены в грамматике за счет наличия двух категорий операндов и операций: слагаемые для операций + и - и множители для операций * и /. Соответствующая программа приведена в алгоритме 7.3.

Полученная грамматика является (косвенно) рекурсивной: конструкция “выражение” определена через “слагаемое”, которое определяется через “множитель”, а его определение по кругу ссылается на “выражение”.

Поэтому и программа содержит вызывающие по кругу друг друга косвенно рекурсивные функции vyrazh(), slag() и mnozh().

Выходом из этого круга служит нерекурсивная ветка в грамматике

множитель ::= число

и соответствующая ветвь оператора if в функции mnozh(), не содержащая рекурсивного вызова функции vyrazh().

Алгоритм 7.3. Вычисление целочисленного выражения

```
/*Вычисление инфиксного выражения без пробелов с целыми    */
/* десятичными числами, операциями +, -, *, / и скобками.  */
/* Метод рекурсивного спуска.                               */

#include <stdio.h>
char c;                 /* Текущий символ входного текста   */
int slag ();
int mnozh ();
int chislo ();

/*    выражение ::= слагаемое [{+ | -} слагаемое]...      */
int vyrazh ()
{  int z, znak;
   c=getchar();
   z = slag();
   while (c=='+' || c=='-')
   {  if (c=='+')  znak=1;  else  znak=-1;
      c=getchar(); z=z+znak*slag();
   }
   return z;                    /* Значение выражения    */
}
/*    слагаемое ::= множитель [{* | /} множитель]...     */
int slag ()
{  int z, d;
   z = mnozh();
   while (c=='*' || c=='/')
   {  if (c=='*')
      { c=getchar();  z=z*mnozh(); }
      else
      {  c=getchar(); d=mnozh();
         if (d!=0)  z=z/d;  else  z=0;        /* Ошибка  */
      }
   }
   return z;                    /* Значение множителя    */
}
/*    множитель ::= число | (выражение)                   */
int mnozh ()
{  int z;
   if (c>='0' && c<='9')  return chislo();
   else if (c=='(')
   {  z=vyrazh();  c=getchar();  return z; }
   else  return 0;                          /* Ошибка  */
}
/*    число     ::= цифра...                              */
int chislo ()
{  int z;
```

```
   z=0;
   while (c>='0' && c<='9')
   {  z=10*z+c-'0'; c=getchar(); }
   return z;                      /* Значение числа         */
}
int main ()
{  printf ("%d \n", vyrazh());
   getch();
}
```

Пример 7.2. Анализ текста - последовательности слов. Пусть входной текст представляет собой последовательность слов, разделенных пробелами, символами "новая строка" и др.

Пример текста: to be or not to beEOF

С этим текстом можно решать разные задачи, например:

а) найти самое длинное слово (такой пример приведен в первой части учебника [67]);

б) подсчитать количество слов;

в) составить словарь слов с указанием, сколько раз в тексте встречается каждое слово (см. раздел 5) и т.п.

Синтаксис (структуру) текста можно описать разными грамматиками и от этого зависит алгоритм анализа текста. Приведем два из возможных вариантов.

1. Рассмотрим текст как последовательность слов. Тогда возможна следующая грамматика для текста:

текст ::= [слово]**. . .**EOF
слово ::= [разделитель]**. . .**[символ_слова]**. . .**
разделитель ::= пробел | новая-строка

Алгоритм чтения и анализа текста для этой грамматики имеет вид:

```
while ((sim=getchar())!=EOF)  /*пока не конец файла  */
{
   /* Пропуск разделителей                            */
   while (sim==' '||sim=='\n')
      sim=getchar();
   /* Чтение символов слова                           */
   while (sim!=' '&& sim!='\n' && sim != EOF)
   {
      sim ==> слово
      sim = getchar();
   }
    . . .        /* Действия, связанные с семантикой */
}
```

2. Представляя текст как последовательность символов, получим другой вариант грамматики для текста:

текст ::= [символ ...] EOF
символ ::= разделитель | символ_слова
разделитель ::= пробел | новая_строка
символ_слова ::= не_разделитель

Алгоритм чтения и анализа текста для этой грамматики будет другим:

```
while ((sim=getchar())!=EOF)  /*пока не конец файла  */
{
   if (sim!=' ' && sim!='\n' && sim!= EOF)
      sim ==> слово
   else
   {
     /* разделитель, т.е. конец слова                */
     . . .     /* Действия связанные с семантикой    */
   }
}
```

Оба варианта алгоритма содержат одинаковые операции (обработку разделителя и символа слова, действия связанные с семантикой текста – смыслом задачи), но по-разному управляют последовательностью их выполнения. Каждый из этих вариантов имеет свои достоинства и недостатки.

Рекурсивный спуск, по сути дела, представляет собой методику построения программ. При его использовании можно написать программу, как метко сказано в одной книге, “почти так же быстро, как мы вообще можем писать”. Другим достоинством является уменьшение вероятности ошибок в программе благодаря соответствию между грамматикой и алгоритмом. Возникающие при этом ошибки обычно бывают довольно простыми и легко исправимыми. Как видно из примера 7.1, в написанную таким способом программу легко вносить изменения.

Недостаток этого метода в том, что написанные таким способом программы получаются громоздкими и выполняются сравнительно медленно из-за большого числа вызовов подпрограмм. Для некоторых видов грамматик этот метод напрямую вообще не работает [15, 75].

В разделе 9 описывается построенный методом рекурсивного спуска компилятор С0, переводящий программу с простого варианта упрощенного языка С на язык ассемблера персонального компьютера IBM PC. Для трансляции выражений в компиляторе С0 используется метод стека с приоритетами, рассмотренный в следующем разделе.

7.2. Трансляция выражений

Рассматриваются некоторые методы обработки выражений в трансляторах, обучающих и других программах. Эти методы полезны для организации более интеллектуального ввода числовых данных в прикладных программах.

Алгоритмы 7.4 - 7.8 сформулированы для бинарных операций, но легко обобщаются на любое число операндов.

7.2.1. Способы записи выражений

Обычно в выражениях знак операции записывается между операндами и иногда требуются скобки:

a + b - *инфиксная запись* (in - внутри).

Польский математик Ян Лукашевич предложил два других способа - записывать знак операции перед операндами или после операндов:

+ a b - *префиксная запись* (прямая польская запись);

a b + - *постфиксная запись* (обратная польская запись).

Функциональная запись f(a, b) фактически является разновидностью префиксной записи, поскольку f можно рассматривать как операцию.

Польская запись является *бесскобочной*, так как позволяет написать любое выражение без скобок, например:

a + (b - 1) / 4 - инфиксная запись;

+ a / - b 1 4 - префиксная запись;

a b 1 - 4 / + - постфиксная запись.

Порядок следования операндов во всех трех способах одинаков, отличается только порядок операций.

В польской записи не требуются скобки, поскольку операции выполняются том порядке, в котором они написаны (если префиксную запись читать справа налево). Поэтому эти способы, особенно постфиксная запись, используются в трансляторах (и микрокалькуляторах).

Постфиксную запись легко транслировать (интерпретировать или компилировать) с помощью стека за один просмотр слева направо. В постфиксной форме можно записывать не только выражения, но и операторы, т.е. всю программу. Она широко используется в трансляторах в качестве промежуточного языка.

7.2.2. Трансляция постфиксного выражения

При интерпретации (вычислении) каждый операнд выражения помещается в стек, а операция выполняется над последними элементами стека (операнды выталкиваются, а результат операции помещается в стек) – алгоритм 7.4.

Алгоритм 7.4. Вычисление (интерпретация) постфиксного выражения

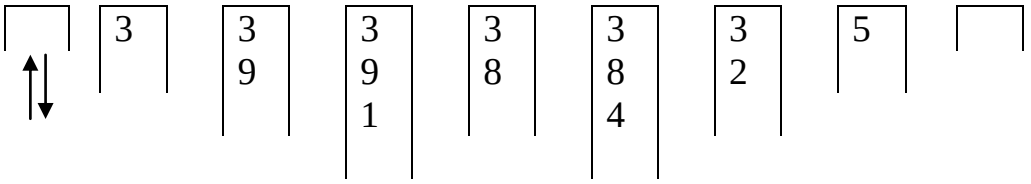
```
Создать пустой Стек;
while (не конец выражения)
{  Чтение s;              /* Текущее слово (операнд | операция) */
    if (s - операнд) Стек <== числовое значение s;
    else                          /* s - операция              */
    {  Стек ==> y;  Стек ==> x;
       Стек <== s(x, y);          /* Операция s над x и y      */
    }
}
Стек ==> Результат;
```

Тест. a=3, b=9. Вход: a b 1 - 4 / + Выход: 5

***Трассировочная таблица*:**

s =	a	b	1	-	4	/	+	
x =				9		8	3	
y =				1		4	2	
Результат =								5

Стек =



Алгоритм компиляции (перевода) постфиксного выражения такой же, как при интерпретации (алг. 7.5): с помощью стека определяются операнды каждой операции. Только вместо выполнения операции в выходную программу помещается ее перевод на другой язык, а в стек записывается имя переменной, которой присваивается результат. Поэтому компилятору не требуются числовые значения операндов: он переписывает тексты операндов из входного выражения в выходную программу.

Пусть, например, выражение переводится в последовательность элементарных присваиваний, содержащих одну арифметическую операцию. Используются вспомогательные переменные вида Ri (R1, R2, ...). После использования промежуточного результата Ri переменная Ri освобождается, и ее можно использовать вновь. Это позволяет экономить переменные.

Примечание. Переменные Ri тоже образуют стек, но, в отличие от использованного в алгоритме 7.5 стека периода трансляции, он функционирует, подобно стеку из алгоритма 7.4, при выполнении транслированной программы, и его можно назвать стеком выполнения.

Алгоритм 7.5. Перевод (компиляция) постфиксного выражения в элементарные присваивания

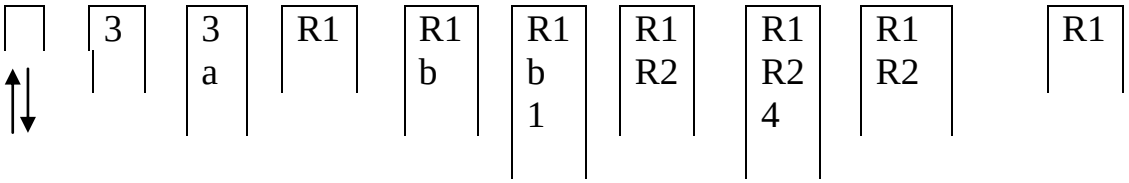
```
i = 1;                    /* Номер очередной переменной вида Ri */
Создать пустой Стек;
while (не конец выражения)
{   Чтение s;             /* Текущее слово (операнд | операция) */
    if (s - операнд) Стек <== s;
    else                                  /* s - операция       */
    {  Стек ==> y;  Стек ==> x;
       if (y имеет вид Rj) i = j;         /* Rj освободится (j<i)*/
       if (x имеет вид Rj) i = j;         /* Rj освободится (j<i)*/
       Вывод "Ri = x s y ;" ;      /* Вместо i,x,s,y - значения */
       Стек <== "Ri";  i++;
    }
}
```

Тест. Вход: 3 a * b 1 - 4 / +
Выход: R1=3*a; R2=b-1; R2=R2/4; R1=R1+R2;

Трассировочная таблица:

s =	3	a	*	b	1	-	4	/	+
x =			3			b		R2	R1
y =			a			1		4	R2
i =	1		2			3		2 3	2 1 2
Вывод:			R1=3*a			R2=b-1		R2=R2/4	R1=R1+R2

Стек =



7.2.3. Трансляция инфиксного выражения

Идею метода *стека с приоритетами* для трансляции выражений предложил Э. Дейкстра в 1960 г. Этот метод позволяет за один просмотр инфиксного выражения слева направо либо перевести его в постфиксную запись, либо интерпретировать его, т. е. вычислить, либо компилировать - перевести на другой язык.

Выражение рассматривается как последовательность *секций*, состоящих из *терма* (операнда), который может отсутствовать, и *ограничителя*:

выражение ::= секция ...
секция ::= [терм] ограничитель
терм ::= число | имя
ограничитель ::= знак-операции | (|) | конец-выражения
конец-выражения ::= ∇

К ограничителям относятся знаки операций, скобки, символ конца выражения и др. Ограничителям присвоены *приоритеты* по старшинству операций (табл. 7.1). Набор операций легко расширять, дополнив таблицу приоритетов.

Для определения порядка выполнения операций используется стек.

Таблица 7.1

Приоритеты ограничителей

Ограничитель	Приоритет
(	0
) конец выражения	1
+ -	2
* /	3

1. Перевод инфиксного выражения в постфиксную запись

Порядок операндов в инфиксном и постфиксном выражениях совпадает. Поэтому каждый операнд (терм) входного выражения сразу переписывается на выход. Ограничители переупорядочиваются с помощью стека. Элемент стека содержит ограничитель (и его приоритет). Каждый ограничитель выталкивает из стека в выходной текст ограничители с большим, чем у него, или равным приоритетом, а затем помещается в стек.

Исключение делается для скобок: открывающая скобка сразу помещается в стек без проверки приоритетов, а закрывающая скобка выталкивает из стека ограничители до соответствующей открывающей скобки, но сама в стек не помещается, а вместо этого выталкивает открывающую скобку без записи в выходной текст (алг. 7.6).

Необходимо, чтобы никакая операция не могла вытолкнуть открывающую скобку, а закрывающая скобка выталкивала все операции. Поэтому скобкам назначают приоритет меньше, чем у операций. Символ конца выражения играет роль закрывающей скобки всего выражения и имеет такой же приоритет.

Алгоритм 7.6. Перевод инфиксного выражения в постфиксное

```
Создать пустой Стек;
while (s != конец-выражения)
{  Чтение s;              /* текущее слово (терм | ограничитель) */
   if (s - терм)    Вывод s;
   else                                  /* s - ограничитель     */
   {  if (s != '(')
         /* Выталкивание более приоритетных ограничителей       */
         while (Стек не пуст && приоритет s <= приоритет Стек)
         {  Стек ==> y;
           Вывод y;
         }
      if (s == ')') Стек ==>;      /* вытолкнуть '(' без вывода */
      else          Стек <== s;
   }
}
```

Тест. Вход: a + (b - 1) / 4 ∇ Выход: a b 1 - 4 / +

***Трассировочная таблица*:**

s =		a		+		(		b		-		1		)				/		4				∇
Выход =		a						b				1		-						4		/		+
Стек =																								
				+		+		+		+		+		+		+		+		+		∇		
						(		(		(						/								
								-																

2. Интерпретация инфиксного выражения

При интерпретации или компиляции инфиксного выражения элемент стека содержит секцию, т.е. терм и ограничитель. Терм каждой секции сразу помещается в стек, а ограничитель сначала выталкивает из стека ограничители с большим, чем у него, или равным приоритетом, а затем тоже помещается в стек.

Исключение делается для скобок: открывающая скобка сразу помещается в стек без выталкивания ограничителей, а закрывающая скобка выталкивает из стека все операции до соответствующей открывающей скобки, но сама в стек не помещается, а вместо этого выталкивает открывающую скобку. Для этого скобкам назначают приоритет меньше, чем у операций.

Символ конца выражения играет роль закрывающей скобки всего выражения и поэтому имеет такой же приоритет.

Выталкивание ограничителя означает реализацию соответствующей ему операции над нужным для нее количеством последних термов стека, удаление из стека этих термов и запись на их место информации о результате операции.

Выталкивание открывающей скобки сводится к продвижению на одну позицию вглубь стека информации о значении заключенного в скобки подвыражения, находящейся в верхушке стека.

Во время интерпретации при выталкивании ограничителя под *реализацией* соответствующей ему операции понимается ее выполнение над операндами из стека. Поэтому в стек помещаются значения (а не тексты) термов (алг. 7.7).

В алгоритме 7.7. Стек.терм и Стек.огр обозначают, соответственно, терм и ограничитель вершины стека.

Алгоритм 7.7. Вычисление (интерпретация) инфиксного выражения

```
Создать пустой Стек;   d=' ';
while (d != конец-выражения)
{   /*          Чтение секции                                          */
    if (d != ')')
    {   t = значение терма; Стек.терм <== t; }
    d = ограничитель;
    if (d != '(')
       /* Выталкивание более приоритетных ограничителей               */
       while (в Стеке есть ограничители
              && приоритет d <= приоритет предпоследнего Стек.огр)
       {   Стек.терм ==> y;
           Стек.терм = операция Стек.огр (Стек.терм, y);
       }
    if (d == ')')
    {  Стек.терм ==> y;  Стек.терм = y; }              /* Вытолкнуть '('  */
    else Стек.огр = d;
}
Стек.терм ==> Результат;
```

Тест. a=3, b=9. Вход: a + (b - 1) / 4 ∇ Выход: 5

***Трассировочная таблица*:**

Секция=	a +	(	b-	1)	/	4∇	
Результат							5
Стек =	3+	3+ (	3+ (9-	3+ (9- 1	3+ (8	3+ 8	3+ 8 /	3+ 8 / 4	3+ 2	5∇	

3. Компиляция инфиксного выражения

При компиляции (переводе) так же, как при интерпретации, с помощью стека определяется порядок выполнения операций и их операнды. Только вместо выполнения операции в выходную программу помещается ее перевод на другой язык, а в стек записывается имя переменной для результата.

Компилятору не требуются значения операндов: он переписывает операнды из входного текста в выходной текст.

Пусть, например, выражение переводится в последовательность элементарных присваиваний, содержащих одну арифметическую операцию (алг. 7.8). Используется минимальное количество вспомогательных переменных вида Ri (R1, R2, ...).

Алгоритм 7.8. Перевод (компиляция) инфиксного выражения в элементарные присваивания

```
d=' '; i=1;                              /* Номер очередной переменной вида Ri */
Создать пустой Стек;
while (d != конец-выражения)
{   /* Чтение секции */
    if (d != ')')
    {   t = терм;  Стек.терм <== t;  }
    d = ограничитель;
    if (d != '(')
        /* Выталкивание более приоритетных ограничителей */
        while (в Стеке есть ограничители
            && приоритет d <= приоритет предпоследнего Стек.огр)
        {    Стек.терм ==> y;
             if (y имеет вид Rj)            i = j;          /*  Rj освободится     */
             if (Стек.терм имеет вид Rj)  i = j;            /*  Rj освободится     */
             Вывод "Ri = Стек.терм Стек.огр y ;" ;          /* Вместо i,
                                                  Стек.терм, Стек.огр, y - значения */
            Стек.терм = "Ri"; i++;
        }
     if (d == ')')
     {  y <== Стек.терм; Стек.терм = y; }                   /* Вытолкнуть '('      */
     else  Стек.огр = d;
}
```

Тест. Вход: a + (b - 1) / 4 ∇

Выход: R1=b-1; R1=R1/4; R1=a+R1;

Трассировочная таблица:

Секция	a+	(	b-	1)	/	4∇		
i =	1			2		1 2	1 2	
Вывод				R1=b-1		R1=R1/4	R1=a+R1	
Стек =		a+	a+ (	a+ (b-	a+ (b- 1	a+ (R1	a+ R1	a+ R1/	a+ R1/ 4	a+ R1	R1∇	

7.3. Задачи

7.1. Представить в двух других формах записи
а) инфиксное выражение 9 - 8 / (a + 1) * 3
б) префиксное выражение - / a 2 * 3 + b 8
в) постфиксное выражение x 1 y + 4 / 5 - *

7.2. Определить результат и составить трассировочную таблицу для
а) перевода постфиксного выражения a 1 - 3 % 4 b * + в элементарные присваивания;
б) перевода инфиксного выражения a+(3*b-7)/4 в постфиксное;
в) вычисления значения инфиксного выражения a+(3*b-7)/4 при a=2 и b=9;
г) перевода инфиксного выражения a+(3*b-7)/4 в элементарные присваивания.

7.3. Входной текст представляет собой выражение из целых чисел с операциями +, -, *, / и скобками и заканчивается знаком "новая строка" или пробелом. Составить программу вычисления значения соответствующего выражения.
Указание: использовать метод стека с приоритетами.
Пример. Вход: 29+3*(540-(100+7)*5)+126/30
Выход: 48

7.4. Входной текст представляет собой выражение из целых чисел с операциями +, -, *, / и скобками и заканчивается знаком ';'. Составить алгоритм получения бинарного дерева, определяющего структуру данного выражения.
Для анализа выражения и определения порядка выполнения операций использовать
а) рекурсивный спуск;
б) метод стека с приоритетами.

7.5. Дано дерево арифметического выражения с целочисленными операндами и операциями сложения, вычитания, умножения и деления. Составить фрагмент программы вычисления значения выражения. Представление дерева выбрать самостоятельно.

Указание: выполнить обход дерева в обратном порядке – снизу вверх, поскольку для выполнения операции необходимо сначала вычислить ее операнды. При таком обходе выражение фактически читается и вычисляется в постфиксной записи.