МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ ИНДИВИДУАЛЬНОГО ЗАДАНИЯ ПО ДИСЦИПЛИНЕ «ТЕОРИЯ АЛГОРИТМОВ»

(для студентов направления подготовки 09.03.02 «Информационные системы и технологии»)

В методических указаниях изложены теоретические основы описания синтаксиса формальных языков, методов разработки порождающих грамматик, грамматик-распознавателей и методов синтаксического анализа конструкций языков программирования. Теоретический материал иллюстрирован примерами, которые направлены на практическое применение теоретических знаний, приведены задания для выполнения работы.

Тема: Формальные языки.

Цель работы: Изучение метаязыков описания синтаксиса формальных языков, методов разработки порождающих грамматик, грамматик-распознавателей и методов синтаксического анализа конструкций языков программирования.

1. Теоретические положения

Теория формальных грамматик и языков – раздел математической лингвистики.

Возникла в 50-е годы 20 века в работах американского лингвиста Н. Хомского. Изначально занимался изучением строения естественных языков. Применяемые им методы могут быть использованы для изучения специализированных языков, в частности, алгоритмических.

По характеру используемого математического аппарата ТФЯ близка к теории алгоритмов и теории конечных автоматов.

Любой язык описывается:

1) алфавитом;

2) синтаксисом – набор правил для построения и описания форм, конструкций, предложений языка;

3) семантикой – смысл, толкование конструкций языка, правила использования синтаксиса.

Предложение – конечная последовательность слов в некотором алфавите.

Грамматика – набор правил, описания синтаксиса языка.

Описание любого формального языка осуществляется обычно на другом языке, называемом метаязыком.

Метаязык может описывать либо синтаксис формального языка (форму конструкций), либо семантику (смысл этих конструкций), либо то и другое вместе.

Для языков программирования наиболее распространенными метаязыками описания синтаксиса являются:

- нормальные формы Бэкуса или форма Бэкуса-Наура (БНФ);
- модифицированные формы Бэкуса-Наура (МБНФ);
- синтаксические диаграммы (СД).

1.1. Основные понятия БНФ

- **Терминальный символ** – символ, состоящий только из букв алфавита описываемого языка. В общем случае символом может быть одна или несколько букв, имеющие вместе определенное значение. Например, ключевое слово END.
- **Нетерминальный символ** – сформулированное на русском или любом другом языке понятие описываемого языка программирования.

Нетерминальные символы заключаются в угловые скобки.

Например: **<программа>**, **<символическое имя>**, **<арифметическое выражение>.**

Нетерминальные символы называют **металингвистическими переменными, переменные метаязыка.**

Специальные метасимволы

- “**::=**” – по определению есть или представляет собой;
- “**|**”– или.

Примеры

<*цифра***>** **::=** *0* **|** *1* **|** *2…***|** *9.*

<*оператор***>** **::= <***оператор присваивания* **> |**

<*оператор цикла***> | <***условный оператор***>**

Для того, чтобы раскрыть понятие языка, обозначаемое нетерминальным символом, используется **правила подстановки (ПП)** или **металингвистические формулы,** являются предложениями метаязыка.

В общем случае ПП:

***U*::=*u*,**

где ***U, u*** – произвольные (конечные) последовательности нетерминальных и терминальных символов.

Обычно, в языках программирования

U – один нетерминальный символ,

u – любая (в том числе и пустая) последовательность нетерминальных и терминальных символов, раскрывающая сущность (возможно не до конца) нетерминального символа, стоящего в левой части подстановки.

Пример 1.

Описание синтаксиса языка целых чисел

- **<***целое***>::= <***цифры***>|<***знак***> <***цифры***>**

- **<*цифры*>::= <*цифра*> | <*цифры*><*цифра*>**
- **<*знак*>::= + | -**

<*цифра*>::= *0/1/2…/9.*

Пример 2.

Грамматика символического имени

<*символическое имя*>::= <*буква*> |

<*символическое имя*> <*буква-цифра*>

<*буква-цифра*>::= < *буква*> | < *цифра*>

- **<*буква*>::= A | …| z**
- **<*цифра*>::=*0*|*1*| … | *9*.**

Во многих случаях для сокращения записи и повышения наглядности описания синтаксиса используют так называемую модифицированную БНФ.

Модификация заключается в добавлении новых символов метаязыка.

Наиболее часто применяют квадратные и фигурные скобки.

Часть предложения, заключенная в квадратные скобки может либо присутствовать, либо отсутствовать.

Пример 3.

- **<*вещ. константа*>::=**
  [<*знак*>]<*целое*>.<*целое*>[*E* [<*знак*>] *целое*>]
- **<*знак*>::= + | -**
- **<*целое*>::=<*цифра*>|<*целое*><*цифра*>**
- **<*цифра*>::=***0* | *1* | … | *9*.**
- Часть предложения, заключенная в фигурные скобки, может либо отсутствовать, либо повторяться любое число раз.

Пример 4

- **<*символическое имя*>::= <*буква*> {<*буква*> | <*цифра*>}**
- **<*буква*>::= A | … | z**
- **<*цифра*>::= 0 | 1 | ... | 9**

Пример 5

- **<*список имен*>::= <*имя*> { , < *имя*> }**
- Число повторений части предложения, заключенного в фигурные скобки, может быть ограничено снизу и сверху подстрочными и надстрочными индексами.
- Это записывается следующим образом: ,
  где m – минимальное, n– максимальное число повторений.
  Например, цепочка символов:

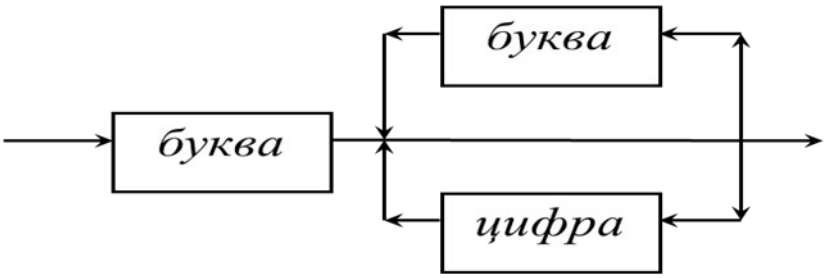
Определяет символическое имя длиной не более 6 символов.

- Третий распространенный способ описания синтаксиса языков программирования – это синтаксические диаграммы, являющиеся графическим изображением БНФ.
- Нетерминальные символы изображаются на СД в прямоугольниках, терминальные в кружках или овалах (для длинных терминальных символов).
- Символы соединяются стрелками, указывающими последовательность чтения символов, образующих цепочку.
- Символ или (“|”) изображается на диаграмме петлей.

Пример 6

Определение символического имени

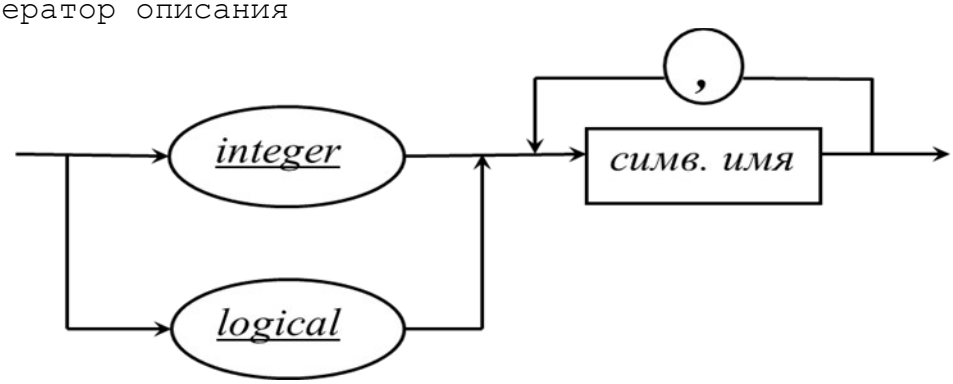
- Символическое имя



Пример 7

Определение синтаксиса оператора описания

Оператор описания



1.2. **Порождающая грамматика (ПГ)**

■ Синтаксис всего языка или его части определяется последовательностью правил подстановки.

■ Одно из этих правил, обозначающее наиболее общее (основное) понятие языка стоит первым,

■ нетерминальный символ, обозначающий это понятие – **начальный символ** грамматики.

■ Упомянутая последовательность правил подстановки называется **порождающей грамматикой,** так как она описывает процедуру получения правильных предложений языка.

■

1.2.1. **Формальное определение ПГ**

ФПГ *G* представляет собой четверку объектов:

где

☐ *N* - множество нетерминальных символов, обозначающих конструкции описываемого языка;

☐ *T* - множество терминальных символов, состоящих из букв алфавита описываемого языка;

☐ начальный символ грамматики, стоит в левой части первого правила подстановки;

☐ *П* – набор правил подстановки.

Таким образом, чтобы описать синтаксис ФЯ, необходимо задать

$$N, T, \sum, П.$$

1.2.2. **Словарь** грамматики:

■ множество цепочек терминальных и нетерминальных символов, включая пустую цепочку.

■ множество цепочек терминальных и нетерминальных символов, исключая пустую цепочку.

Пошаговый процесс получения конструкций языка по ФПГ называется **выводом** или **порождением.**

Вывод заканчивается, если все символы терминальные.

Пример 8. Вывод по грамматике примера 1 цепочки символов - **247.**

<*целое***>::= <***знак***> <***цифры***>**
- <*цифры***>**
- <*цифры***> <***цифра***>**
- <*цифры***> <***цифра***> <***цифра***>**
- <*цифра***> <***цифра***> <***цифра***>**
- 2 <*цифра***> <***цифра***>**
- 24 <*цифра***>**
- 247.

■ Примеры вывода символических имён по грамматике 2:

1) **<***символическое имя***> => <***символическое имя***> < *буква-цифра***> => <***символическое имя***> < *буква-цифра***> < *буква-цифра***> => <***буква***> < *буква-цифра***> < *буква-цифра***> => X< *буква-цифра***> < *буква-цифра***> => X <**буква**> <***буква-цифра* **> => XY< *буква-цифра* > => XY<**цифра**> => XY9**

2) **<***символическое имя***> => <***буква***> => A**

■ В этом примере использован один очень простой прием – рекурсивное определение. В первом правиле подстановки **<***символическое имя***>** определяется вначале простейшим образом как **<***буква***>**, а потом, после разделителя **|** ("или"), нетерминальный символ **<***символическое имя***>** используется с определенным значением.

■ После того, как **<***символическое имя***>** получило значение **<***буква***> <***буква-цифра***>**, на следующем шаге оно может быть определено как

<*буква***> < *буква-цифра* > < *буква-цифра***>**

и так далее.

■ *Замечание:*

■ На любом шаге вывода можно по-разному раскрывать нетерминальные символы, если в правых частях правил подстановки присутствуют разделители "или".

■ Использование рекурсивного определения нетерминальных символов позволяет получить сколь угодно длинное предложение языка, так как рекурсия естественным образом обеспечивает повторяющийся (циклический) процесс.

■ Вообще, если необходимо описать синтаксис сколь угодно длинной последовательности объектов, следует применять рекурсию.

<*последовательность объектов***>** **::=** **<***объект***>** **|** **<***последовательность объектов***>** **<** *объект>*

■ Пример описания синтаксиса простого языка программирования.

■ В этом описании можно найти все особенности использования БНФ, в частности описания синтаксиса арифметических и логических выражений, являющееся уже классическим.

■ Пример. Грамматика языка программирования

■ Вначале зададим некоторое неформальное описание этого языка.

■ Язык позволяет оперировать с данными целого (**integer**) и логического типов (**logical**).

■ Логические значения: **true**, **false**.

■ Данное может быть константой, переменной или одномерным массивом целых чисел.

■ Одномерный массив: **array** имя **[n:m]**, где **n,m** – целые константы, определяющие нижнюю и верхнюю границы изменения индекса.

■ Элемент массива записывается в виде:
имя **[**индексное выражение**]**.

■ Операторы языка:

☐ оператор присваивания: **x:=E***;*

☐ условный оператор:
if условие **then** оператор **else** оператор **fi** или_
if условие **then** оператор **fi** *;*

☐ оператор цикла: **while** условие **do** оператор **od;**

☐ пустой оператор.

1.2.3. Формальное описание синтаксиса языка в БНФ

■ **<***программа***>::=** **<***описания***>** **<***оператор***>**

■ **<***описания***>** **::=** **<***оператор описания***>;|**
<*описания***>** **<***оператор описания***>;**

■ **<***оператор описания***>::=** **<***оператор описания типа***>** **|** **<***оператор описания массива***>**

■ **<***оператор описания типа***>::=**
<*тип***>** **<***список типа***>**

■ **<***тип* **>::=** **integer**|**logical**

■ **<***список типа***>::=** **<***имя***>** **|**
<*список типа***>,<***имя***>**

■ **<***оператор описания массива***>::=**
array **<***список массивов***>**

■ **<***список массивов***>::=**
<*элемент списка***>** **|** **<***список массивов***>,** **<***элемент списка***>**

■ **<***элемент списка***>::=**
<*имя***>[<***целое***>:<***целое***>]**

■ **<***оператор***>::=** **<***оператор присваивания***>** **|** **<***условный оператор***>** **|** **<***оператор цикла***>** **|**
<*пустой оператор***>** **|**

<оператор> ; <оператор>

- <оператор присваивания>::=
  <переменная> := <выражение>
- <условный оператор>::= **if** < логическое выражение> **then** <оператор> **fi**|
  if <логическое выражение> **then** <оператор> **else** <оператор> **fi**
- <оператор цикла>::= **while** < логическое выражение> **do** <оператор> **od**
- <пустой оператор>::=
- <переменная>::= < простая переменная >|
  <переменная с индексом>
- < простая переменная >::= <имя>
- <переменная с индексом>::=
  <имя> [< арифметическое выражение>]
- <выражение>::= < арифметическое выражение > | < логическое выражение >
- <арифметическое выражение> ::= <слагаемое> | + <слагаемое> |
  - <слагаемое> | <арифметическое выражение> + <слагаемое> |
  <арифметическое выражение> -
  < слагаемое >
- <слагаемое>::= <множитель> | <множитель> * <слагаемое> |
  <множитель> / <слагаемое>
- <множитель>::= <целое> | <переменная>|
  (<арифметическое выражение>)|
  <множитель> ** <множитель>
- <логическое выражение> ::=
  <логическое слагаемое> | <логическое слагаемое> <логическое выражение>
- <логическое слагаемое>::=
  <логический множитель> |
  <логический множитель> <логическое слагаемое>
- <логический множитель>::= <логическая константа> | <простая переменная> |
  <арифметическое выражение> <операция отношения> <арифметическое выражение> | (<логическое выражение>) |
  <логический множитель>
- <логическая константа>::= <**true**> | < **false**>
- <операция отношения>::= |=|>|<| | |
- <имя>::= <буква> | <имя> <буква> | <имя> <цифра>
- <буква>::= A|B|…|z

<цифра>::= 0|1|…|9

1.3. Классификация языков по Хомскому

В основе этой классификации лежит форма левой и правой частей правила подстановки

U::=u.

По Хомскому языки делятся на 4 класса с номерами 0,1,2,3, причем каждый класс с большим номером является подмножеством любого класса с меньшим номером.

Класс 0. Грамматики с фразовой структурой

Правило подстановки имеет вид:

U::=u,

где **U** – произвольная непустая последовательность (цепочка) терминальных и нетерминальных символов;

u - произвольная (возможно и пустая) последовательность терминальных и нетерминальных символов.

Класс 0 является наиболее мощным, языки этого класса могут служить моделью естественных языков.

Класс 1. Контекстно-зависимые грамматики

Правило подстановки имеет вид:

x U y ::= x u y,

где **U** – нетерминальный символ;

x, u, y – произвольные последовательность (цепочка) терминальных и нетерминальных символов.

Смысл этого правила в том, что замена **U** на **u** осуществляется только в контексте **x...y.**

Класс 2. Контекстно-свободные грамматики

Правило подстановки имеет вид:

U::= u,

где **U** – ровно один нетерминальный символ;

u - произвольная цепочка терминальных и нетерминальных символов.

Грамматика типа 2 отличается от грамматики типа 1 тем, что замена **U** на **u** осуществляется в любом контексте.

Грамматика типа 2 обычно используется для описания синтаксиса языков программирования.

Класс 3. Автоматные (регулярные) грамматики

Правило подстановки имеет вид:

U::= t, U::= t n

где **t** – ровно один терминальный символ;

n – ровно один нетерминальный символ.

Грамматика типа 3 используется для описания синтаксиса простых языков программирования (узкоспециализированных или подмножеств языков программирования).

■ Следует иметь в виду, что если хотя бы одно правило подстановки в грамматике относится к более высокому классу (с меньшим номером), чем остальные, то и вся грамматика относится к этому классу.

■ Наибольший интерес в теории языков программирования представляют контекстно-свободные и автоматные грамматики. Одна из причин этого - сравнительная простота распознавания конструкций этих языков, что используется для разработки трансляторов.

1.4. Синтаксический анализ языковых конструкций

■ Синтаксический анализ языковых конструкций представляет собой задачу, противоположную задаче порождения (вывода).

■ Задача синтаксического анализа (задача разбора) формулируется следующим образом: определить, соответствует ли заданная конструкция некоторого языка грамматике этого языка или

другими словами, является ли заданная конструкция правильным (не содержащим грамматических ошибок) предложением языка.

■ Задача разбора имеет широкое практическое применение, каждый транслятор языка программирования имеет в своём составе блок синтаксического анализа.

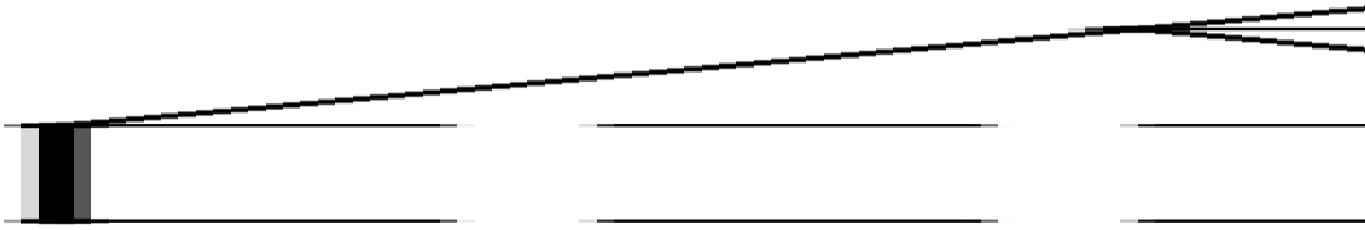
Прежде чем рассматривать алгоритмы синтаксического анализа, введём понятие **синтаксического дерева**, как графического способа изображения вывода конкретного предложения языка.

1.4.1. Дерево

■ представляет собой иерархическую структуру, состоящую из узлов разных уровней;

■ каждый из узлов некоторого уровня (кроме последнего) связан с несколькими узлами следующего уровня и не более, чем с одним узлом предыдущего уровня.

На I уровне может быть только один узел, называемый корнем. Узлы последнего уровня называются листьями. Куст узла – множество подчиненных ему узлов нижних уровней.

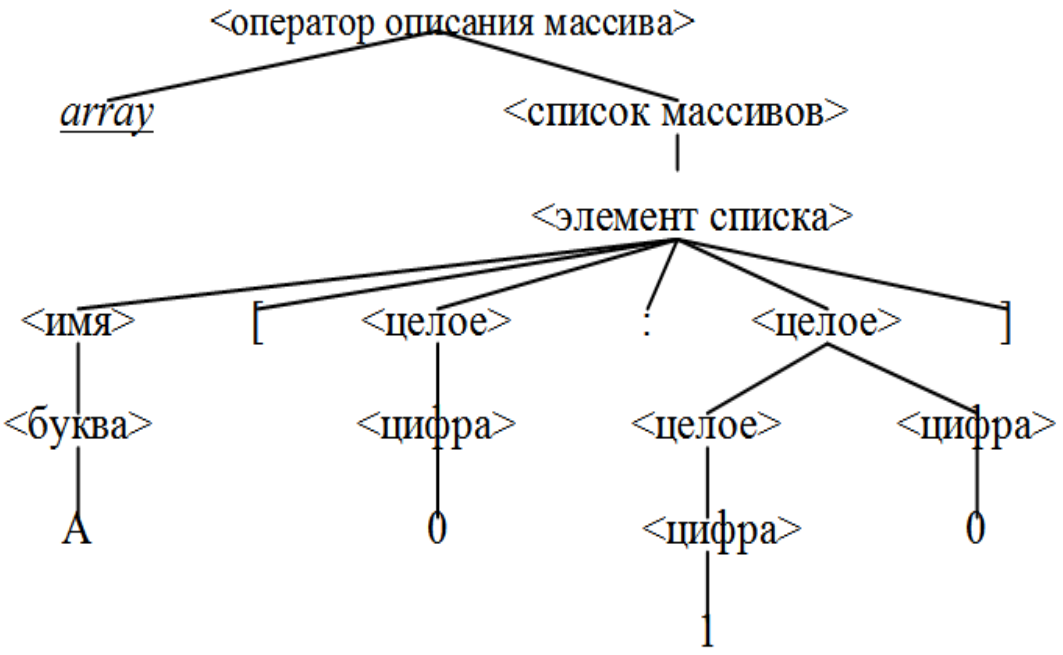


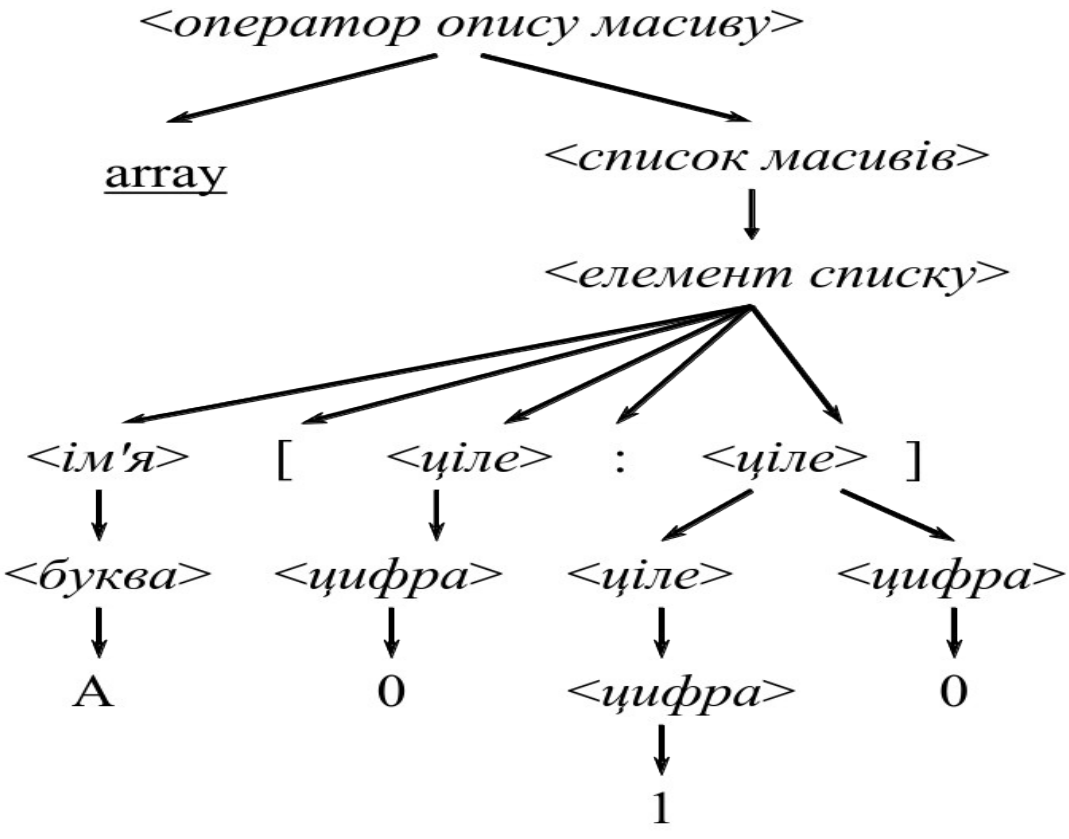
■ В синтаксических деревьях узлы представляют собой нетерминальные и терминальные символы,

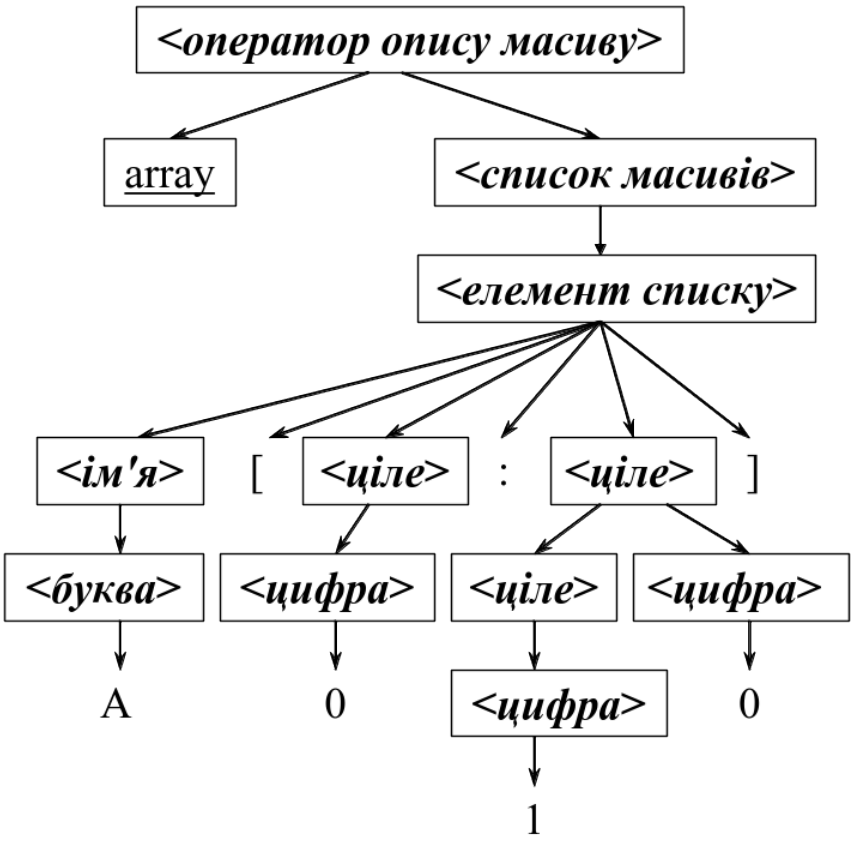
■ листья являются терминальными символами, а остальные узлы – нетерминальными.

■ Корень дерева – начальный символ грамматики. Куст представляет собой правую часть правила подстановки для данного нетерминального символа.

Пример 1. Синтаксическое дерево для описания массива **array A[0:10]**

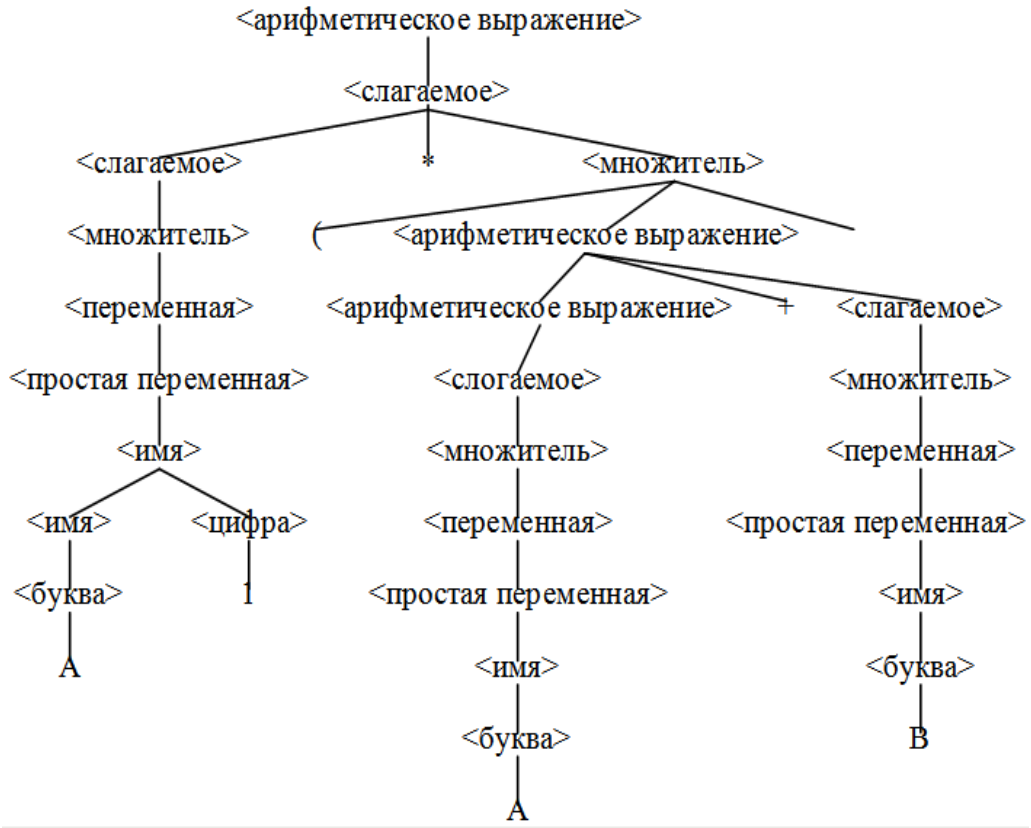

<оператор описания массива>
array
<список массивов>
<элемент списка>
<имя>
[
<целое>
:
<целое>
]
<буква>
<цифра>
<целое>
<цифра>
A
0
<цифра>
0
1


<оператор опису масиву>
array
<список масивів>
<елемент списку>
<ім'я>
[
<ціле>
:
<ціле>
]
<буква>
<цифра>
<ціле>
<цифра>
A
0
<цифра>
0
1



■ Последовательность листьев этого дерева, читаемая слева направо, представляет собой предложение **array A[0:10]**.

■ Пример 2. Синтаксическое дерево для арифметического выражения **A1*(A+B)**



■ Достоинством синтаксических деревьев является наглядность описания синтаксиса как самого предложения, так и его отдельных составляющих.

■ С другой стороны, для реальных конструкций языков программирования дерево оказывается очень громоздким.

■ Поэтому, если дерево используется для получения наглядности описания, то, либо опускаются некоторые промежуточные уровни, несущественные для этого описания, либо оно составляется для простых предложений.

■ Идея построения деревьев широко используется в трансляторах, так как существуют эффективные методы отображения деревьев в памяти ЭВМ.

■ Синтаксический анализ предложений, записанных на языках программирования, фактически сводится к построению синтаксических деревьев различными методами.

■ Существуют два базовых алгоритма синтаксического анализа (решения задачи разбора):

■ нисходящий разбор (развёртка);

■ восходящий разбор (свёртка).

■ При нисходящем разборе синтаксическое дерево строится от корня к листьям.

■ Отличительная черта – целенаправленность, так как, отправляясь от нетерминального символа, мы стремимся найти такую подстановку, которая привела бы к части требуемой цепочки терминальных символов.

■ В простейших случаях синтаксический анализатор (распознаватель) пытается достичь этого путём направленного перебора различных вариантов.

■ Распознаватель рассматривает дерево сверху вниз и слева направо и выбирает первый нетерминальный символ, не имеющий подчиненных узлов (являющийся “листом”).

■ В списке правил подстановки отыскивается правило (или набор правил), которое в левой части содержит этот нетерминальный символ, а в правой части (в правых частях) – очередные (считая слева направо) символы анализируемого предложения.

■ Если такое правило есть, дерево наращивается и описанный процесс повторяется. Если правило не найдено, распознаватель возвращается на один или несколько шагов назад, пытаясь изменить выбор сделанный ранее.

■ Процесс разбора заканчивается в одном из двух случаях:

■ построено дерево, все листья которого являются терминальными символами, и при чтении слева направо образуют анализируемое предложение. В этом случае результат распознавания положителен, то есть синтаксис рассматриваемого предложения соответствует грамматике языка;

■ распознаватель переработал все возможные варианты последовательностей подстановок, но так и не пришёл к дереву, описанному в 1.

■ Это означает, что анализируемое предложение не принадлежит данному языку (то есть содержит ошибки).

■ Следует подчеркнуть, что ошибка считается выявленной только в том случае, когда рассмотрены все допустимые варианты подстановки.

Пример. Провести нисходящий синтаксический разбор числа +2.4 по грамматике

1) **<**константа**> ::= <**КФТ**> | <**знак**> <**КФТ**>**
2) **<**КФТ**> ::= .<**целое**> | <**целое**>. | <**целое**>.<**целое**>**
3) **<**целое**> ::= <**цифра**> | <**цифра**> <**целое**>**
4) **<**знак**> ::= + | –**
5) **<**цифра**> ::= *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9***

■ Процесс разбора покажем в виде последовательности шагов, на каждом из которых строится какая-то часть дерева.

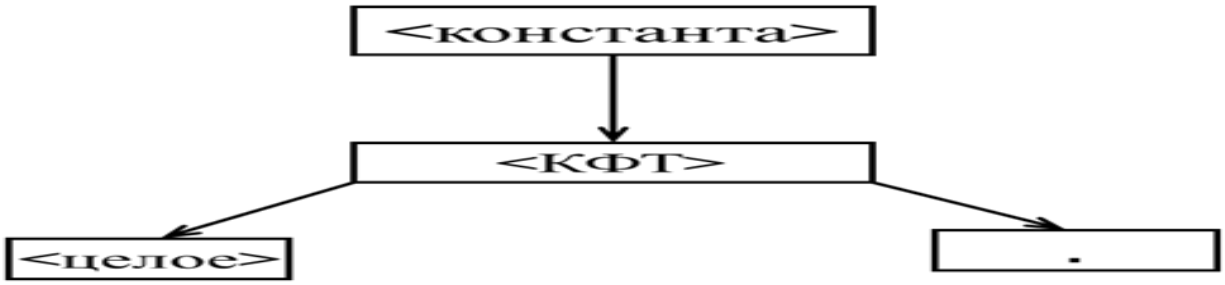
■ Шаг 1. Нетерминальный символ **<**константа**>** – корень дерева.

■ Шаг 2. Поскольку выражение 1) не содержит в правой части терминальных символов, выбираем первую слева цепочку символов:

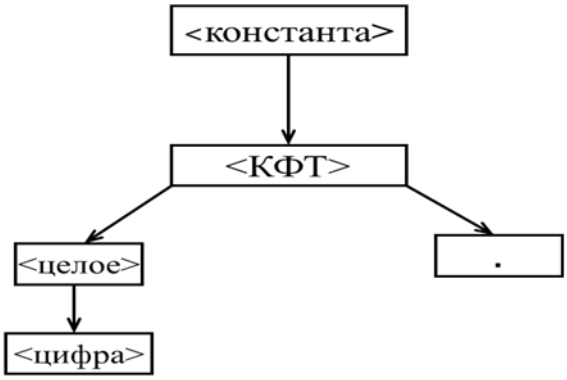
■ **<**константа**> ::= <**КФТ**>.**

■ Получаем

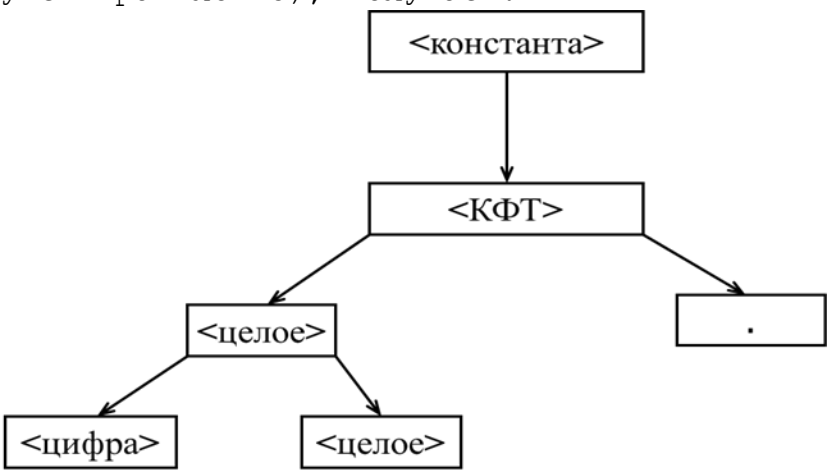
Шаг 3. В правиле **2)** первая цепочка начинается с терминального символа “.”, а так как первый символ анализируемого предложения – “+”, эта цепочка не подходит. Выбираем следующую



- Шаг 4. В правиле 3) выбираем первую цепочку <цифра>:



- Шаг 5. Для определения нетерминального символа <цифра> по правилу 5) есть 10 терминальных символов, но ни один из них не совпадает с “+”.
- Поэтому осуществляем возврат на один шаг назад и, снова пользуясь правилом 3), получаем:

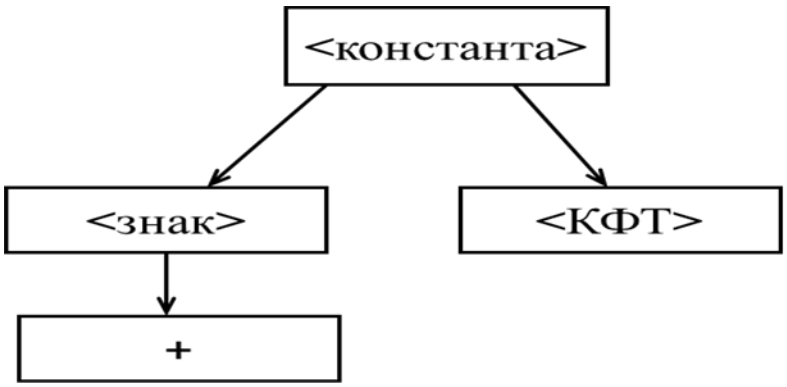


- Шаг 6. Из двух ветвей дерева выбираем, согласно алгоритму, левую и снова пытаемся применить правило 5).
- Так как невозможно, возвращаемся назад к дереву, полученному на шаге 2 (поскольку попытки применения правила 3) исчерпаны.
- В правиле 2) выбираем цепочку <целое>.<целое>. Получаем:

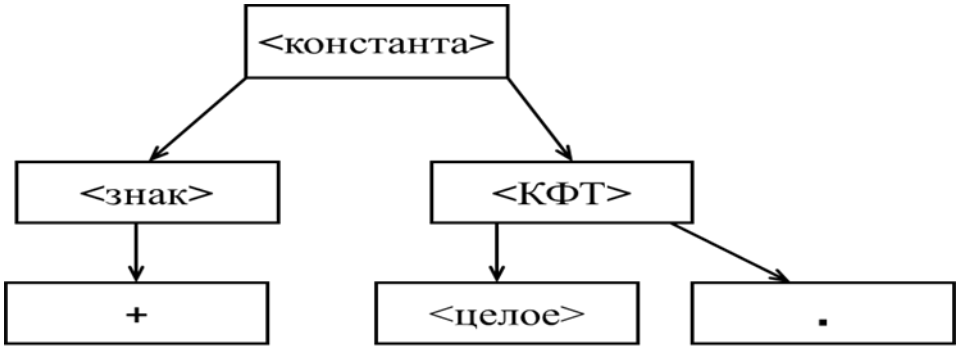
На этом пути мы, очевидно, сделаем ещё 2 шага, прежде чем убедимся в его ложности.

Шаг 9. Возвращается к правилу 1) и выбираем для подстановки вторую цепочку **<**знак**><**КФТ**>**

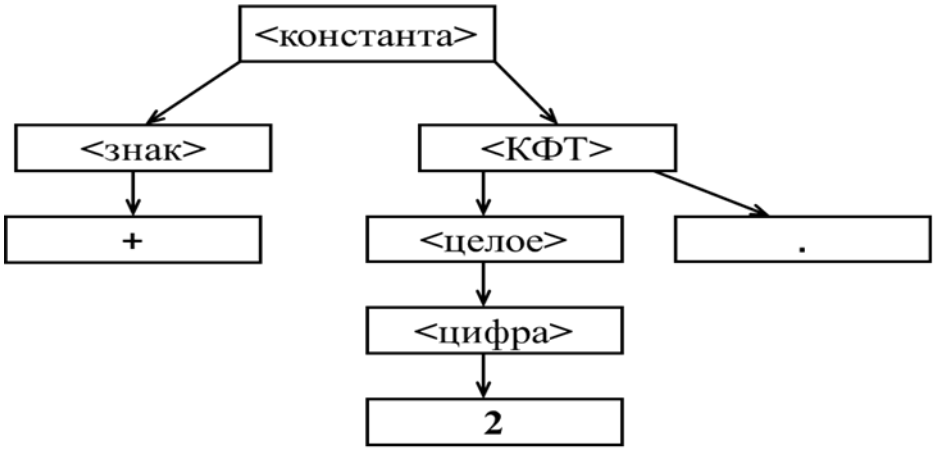
- Шаг 10. Применяем правило 4) к нетерминальному символу <знак>. Получаем.



- Полученный терминальный символ совпадает с первым символом анализируемого предложения, следовательно, ветвь полученного дерева правильна.
- Далее пытаемся получить следующий символ –"2".
- Шаг 11. Раскрываем нетерминальный символ <КФТ> по правилу 2), при этом выбираем вторую цепочку, так как в первой символ "." не совпадает с символом "2".



- Шаги 12, 13. Применяем правила 3) и 5).



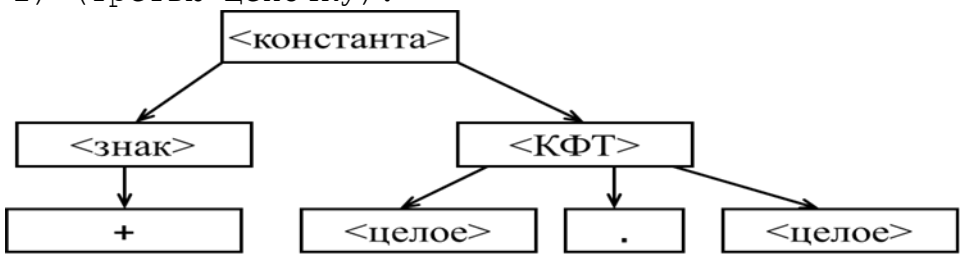
■ Получили сразу два терминальных символа анализируемого предложения “**2**” и “.”. Однако дерево уже построено, так как все листья являются терминальными символами. Если бы все варианты подстановок были уже испытаны, можно было бы сделать вывод, что в предложении +2.4 допущена ошибка. Поскольку это не так, делаем два шага назад, возвращаясь к дереву, полученному на Шаге 11.

■ Шаг 14. Снова применяем правило 3), но его вторую цепочку.

Очевидно, что на этом пути мы сделаем ещё 3 шага, прежде чем зайдём в тупик.

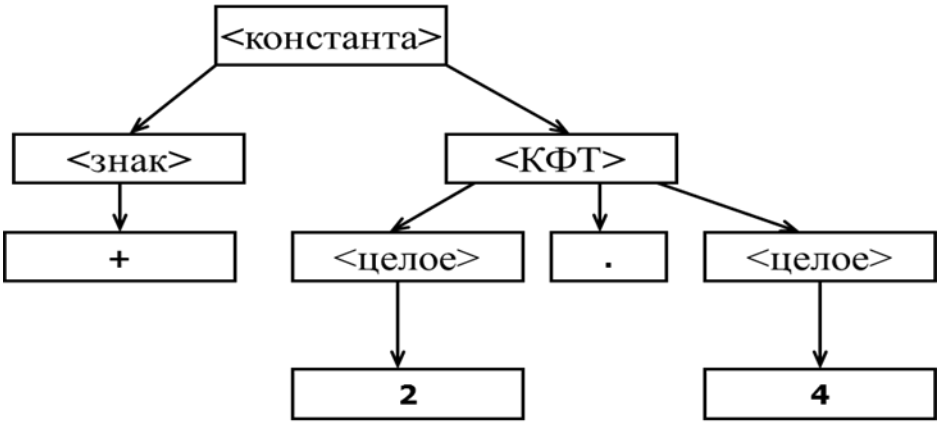
Пропускаем эти три шага и возвращаемся, наконец, к дереву, полученному на шаге 10.

☐ Шаг 18. Применяем к символу <КФТ> на этом дереве правило 2) (третью цепочку).



☐ Шаги 19, 20, 21, 22.

☐ Раскрываем последовательно левый и правый символы <целое>



Листья этого дерева (слева направо) образуют исходное предложение +2.4, следовательно оно соответствует заданной грамматике, т.е. не содержит ошибок.

Всегда следует иметь ввиду, что только неукоснительное, точное следование алгоритму нисходящего разбора позволяет гарантировать правильность разбора, особенно для сложных грамматик. Попытки пропустить несколько шагов как очевидные или сделать их “в уме” часто приводят к ошибкам.

■ Примечание. Описанный алгоритм нисходящего разбора неприменим, если грамматика является леворекурсивной, то есть содержит правила подстановки вида

U ::= Uu,

поскольку при этом попытки раскрыть самый левый нетерминальный символ приведут к бесконечному порождению нетерминального символа **U.** В таком случае необходимо видоизменить алгоритм, сделав его, например, правосторонним, или изменить грамматику.

1.4.2. Синтаксический анализ конструкций языков программирования

■ **В алгоритме нисходящего разбора не обязательно строить дерево. На каждом шаге можно записывать цепочку нетерминальных и терминальных символов, полученных после очередной подстановки.**

■ **При восходящем разборе дерево строится от листьев к корню, то есть алгоритм, отправляясь от заданной строки, пытается, применяя правила подстановки справа налево, привести её к начальному символу грамматики.**

■ Часть строки, которую можно привести к не терминальному символу, называется фразой.

■ Если приведение осуществляется применением одного правила подстановки, фраза называется непосредственно приводимой.

■ Самая левая непосредственно приводимая фраза называется основой.

■ Алгоритм восходящего разбора

В исходном предложении отыскивается основа (то есть просмотр, как и в нисходящем разборе, идет слева направо) и приводится к нетерминальному символу.

■ Эта операция применяется до тех пор, пока либо получен единственный символ, являющийся начальным символом грамматики (в

этом случае разбор заканчивается успешно), либо в полученной цепочке не может быть найдена фраза.

■ В этом случае делается возврат на один или несколько шагов назад и выбирается другая основа.

■ Если все возможные варианты перебраны, а корень дерева так и не получен, делается вывод о наличии ошибки в разбираемом предложении, и алгоритм прекращает работу.

■ Таким образом, восходящий разбор представляет собой перебор вариантов, однако не целенаправленный, так как нет возможности отбрасывать заведомо неправильные подстановки, как это имеет место в нисходящем разборе (цепочка ".<целое>" в правиле 2), неподходящие цифры в правиле 5) в предыдущем примере.

■ Процесс восходящего разбора записывают либо в виде построения дерева от листьев к корню, либо в виде последовательности цепочек терминальных и нетерминальных символов, начинающейся с анализируемого предложения, заканчивающейся (если предложение не содержит ошибок) начальным символом грамматики.

■ Пример. Провести восходящий синтаксический разбор выражения 5*I+J по грамматике

■ 1) <выражение> ::= <слагаемое> | <слагаемое>+<имя> | <слагаемое>-<имя> | <слагаемое>+<целое> | <слагаемое>-<целое>

■ 2) <слагаемое>::= <имя> | <целое> | <целое>*<имя>

■ 3) <целое> ::= <цифра> | <целое><цифра>

■ 4) <имя> ::= *I* | *J* | *K* | *L* | *M* | *N*

■ 5) <цифра>::= *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

■ Разбор представим в виде последовательности шагов, на каждом из которых будет получена цепочка символов, непосредственно приводимые фразы будем подчёркивать. Справа в скобках указаны номера правил, по которым произведена подстановка.

■ Шаг 1. Исходное предложение.

■ Шаг 2. <цифра>*I+J 5)

■ Шаг 3. <целое>*I+J 3)

■ Шаг 4. <слагаемое>*I+J 2)

■ Шаг 5. <выражение>*I+J 1)

■ Шаг 6. <выражение>*<имя>+J 4)

■ Шаг 7.<выражение>*<слагаемое>+J 2)

■ Шаг 8. <выражение>*<выражение>+J 1)

■ Шаг9.<выражение>*<выражение>+<имя> 4)

■ Шаг10. <выражение>*<выражение>

■ +<слагаемое> 2)

■ Шаг11.<выражение>*<выражение>+

■ <выражение> 1)

■ В полученной цепочке нет фразы. Изменить выбор непосредственно приводимой фразы можно, вернувшись к цепочке, полученной на шаге 7:

■ <выражение>*<слагаемое>+J.

■ Шаг 12. <выражение>*<слагаемое>+

■ <имя> 4)

■ Шаг 13. <выражение>*<выражение> 1)

■ В полученной цепочке нет фразы.

- Возвращаемся к шагу 12_ <выражение>*<слагаемое>+<имя>.
- Шаг 14. выражение>*<слагаемое>+
- <слагаемое> 2)
- Шаг 15. <выражение>*<выражение>+
- <слагаемое> 1)
- Получили ту же цепочку, что и на шаге 10. Возвращаемся к шагу 14, сделаем один ложный шаг, затем к шагу 6: <выражение>*<имя>+J.
- Шаг 17. <выражение>*<имя>+<имя> 4)
- Шаг 18. <выражение>*<слагаемое>+
- <имя> 2)
- Возвращаемся к шагу 17.
- Шаг 19. <выражение>*<имя>+
- <слагаемое> 2)
- Шаг 21. <выражение>*I+<имя> 4)
- Возвращаемся к шагу 4: <слагаемое>*I+J
- Шаг 25. <слагаемое>*<имя>+J 4)
- Далее проделаем ещё 18 ложных шагов, после чего вернёмся к шагу 3: <целое>*I+J.
- Шаг 44. <целое>*<имя>+J 4)
- Шаг 45. <слагаемое>+J 2)
- Шаг 46. <выражение>+J 1)
- …

Шаг 49. <выражение>+<выражение> 1)
Возвращаемся к шагу 45.
Шаг 50. <слагаемое>+<имя> 4)
Шаг 51. <выражение>

- Главным недостатком как нисходящего, так и восходящего алгоритмов разбора является большое количество шагов, что определяется переборным характером алгоритмов.

2. Требования к выполнению работы

Задана конструкция языка программирования. Необходимо:

- разработать порождающую грамматику. Для описания правил подстановки использовать:
  - Формы Бэкуса-Наура (нечетные варианты);
  - Модифицированные формы Бэкуса-Наура (четные варианты);
- С помощью разработанной порождающей грамматики выполнить вывод конструкций для фрагментов программы, представляющих заданные конструкции.
- Выполнить синтаксический анализ заданных конструкций, используя алгоритм:
  - Нисходящего разбора (нечетные варианты);
  - Восходящего разбора (четные варианты);.

Примечание:

1. Фрагменты программ, состоящие из вариантов использования заданных конструкций языка придумать самостоятельно. Как минимум по одному фрагменту для корректного использования конструкции и с ошибкой.

2. Неформальное описание заданных конструкций языка найти самостоятельно.

3. Варианты заданий

Группа а

1. Директивы препроцессора.
2. Описание функций в языке С.
3. Определение констант в языке Паскаль.
4. Описание переменных в языке Бейсик.
5. Инициализация строковых значений языке С.
6. Логические выражения языке С.
7. Вывод строк языке С.
8. Описание переменных в языке Паскаль.
9. Ввод с клавиатуры в языке Паскаль.
10. Условный оператор в языке С.
11. Оператор цикла FOR Паскаль.
12. Оператор-переключатель в языке Бейсик.
13. Операторы цикла с пост- и предусловием в языке С.
14. Описание структур в языке Паскаль.
15. Описание пользовательских типов в языке С.
16. Описание функции main в языке С.
17. Функции работы с динамической памятью в языке Паскаль.
18. Функции работы с файлами в языке Бейсик.
19. Описание и вызов функций, аргументом которых является одномерный массив, в языке С.
20. Функции управления цветом и заливкой в графическом режиме (язык С).
21. Описание класса в языке С++.
22. Описание переменных в языке С.
23. Определение констант в языке Бейсик.

Группа б

1. Функции вывода графических фигур (язык С).
2. Описание производного класса в языке С++.
3. Описание переменных в языке Паскаль.
4. Описание функций и процедур в языке Бейсик.
5. Инициализация массивов в языке С.
6. Арифметические выражения языке С.
7. Вывод на экран в языке С++.
8. Вывод строк языке С.
9. Условный оператор в языке Паскаль.
10. Оператор цикла FOR в языке С.
11. Инициализация строковых значений языке С.
12. Оператор-переключатель в языке Паскаль.
13. Операторы цикла с пост- и предусловием в языке Бейсик.
14. Описание структур в языке С.
15. Описание пользовательских типов в Паскаль.
16. Функции работы с динамической памятью в языке С.
17. Функции работы с файлами в языке Паскаль.
18. Описание и вызов функций, аргументом которых является двумерный массив, в языке С.
19. Функции обработки нажатия клавиш в языке С.
20. Перегрузка операций в языке С++.
21. Определение констант в языке С.
22. Описание функций и процедур в языке Паскаль.
23. Логические выражения языке С.

Группа в

1. Вывод на экран в языке С.
2. Ввод с клавиатуры в языке С++.
3. Условный оператор в языке Бейсик.

4. Функции управления курсором и оконным выводом в текстовом режиме (язык С).
5. Оператор-переключатель в языке Паскаль.
6. Описание и вызов функций, аргументом которых является одномерный массив, в языке С.
7. Операторы цикла с пост- и предусловием в языке Паскаль.
8. Операторы для использования динамической памяти в языке С++.
9. Функции работы с файлами в языке С.
10. Оператор цикла FOR в языке С.
11. Функции управления курсором и оконным выводом в текстовом режиме (язык С).
12. Вывод на экран в языке Паскаль.
13. Описание пользовательских типов в языке С.
14. Оператор-переключатель в языке С.
15. Определение динамических массивов в языке Бейсик.
16. Ввод с клавиатуры в языке С.
17. Вывод на экран в языке Паскаль.
18. Оператор цикла FOR Бейсик.
19. Описание пользовательских типов в Паскаль.
20. Оператор-переключатель в языке Бэйсик.

Литература:

1. Алферова З.В. Теория алгоритмов : учебное пособие для вузов / З. В. Алферова ; З.В. Алферова. - М. : Статистика, 1973. - 164с. : черт.
2. Карпов Ю.Г. Теория автоматов : учебник для вузов / Ю. Г. Карпов ; Ю.Г. Карпов. - СПб. : Питер, 2003. - 208с. : ил. - (Учебник для вузов). - ISBN 5-318-00537-3.
3. Петер Р. Рекурсивные функции / Р. Петер. - 1954.
4. Успенский В.А. Теория алгоритмов: основные открытия и приложения / В. А. Успенский. - 1987.
5. Теория алгоритмов и програм : сборник научных трудов / Латвийский государственный университет им. П. Стучки. Вычичлительный центр ; Латв. гос. ун-т им. П. Стучки ; редкол.:Я.М. Барздинь (отв. ред.) и др. - Рига : ЛГУ, 1986. - 194с. : ил.
6. Вычислительные системы : сборник научных трудов / АН СССР. Сибирское отделение ; АН СССР, Сиб. отд-ние, Ин-т математики ; редкол.: Ю.Г.Косарев и др. - Новосибирск, 1961-. [Вып.]129 : Теория алгоритмов и ее приложение / науч. ред. Ю.Л. Ершов, С.С. Гончаров. - 1989. - 198с
7. Трахтенброт, Б.А. Алгоритмы и вычислительные автоматы / Б. А. Трахтенброт ; Б.А. Трахтенброт. - М. : Сов. радио, 1974. - 200с. : ил.
8. Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов. М.:Мир, 1981.
9. Ахо, А.В. Построение и анализ вычислительных алгоритмов / А. В. Ахо, Хопкрофт Дж., Ульман Дж.Д. ; А. Ахо, Дж. Хопкрофт. Дж.Д. Ульман ; пер. с англ. А.О. Слисенко ; под ред. Ю.В. Матиясевича. - М. : Мир, 1979. - 536с. : ил.
10. Ахо, А.В. Структуры данных и алгоритмы / А. В. Ахо, Хопкрофт Дж.Э., Ульман Дж.Д. ; А.В. Ахо, Дж.Э. Хопкрофт, Дж.Д. Ульман ; пер. с англ., ред. А.А. Минько. - М. ; СПб. ; К. : Вильямс, 2007. - 400с. : ил. –