Кафедра «Технологии программирования»

МЕТОДИЧЕСКОЕ ПОСОБИЕ ДЛЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ ПО ДИСЦИПЛИНЕ

"Системное программное обеспечение"

Ташкент 2008

Авторы: Назиров Ш.А., Кабулов Р.В., Урынбаев С.К. Методическое пособие для практических занятий по дисциплине «СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ»

Данное методическое пособие предназначено для студентов специальности информационные технологии изучающих предмет «Системное программное обеспечение».

Цель пособия – закрепить знания, полученные при изучении теоретической части курсов и получить практические навыки разработки формальных грамматик языков программирования.

Методическое пособие соответствует программе курса «Системное программное обеспечение», целью которой является познакомить студента с основным приемами, необходимыми для решения приведенных задач.

№	Темы	Часы
1	Языки и грамматики.	2
2	Формальные грамматики	2
3	Регулярные языки и грамматики	2
4	КС – Грамматики	2
5	LL грамматики	2
6	LR - грамматики	2
7	Генерация выходного текста.	2
8	Элементы теории перевода	2
	Всего	16

Рецензенты:

доцент НУ Уз
кандидат физика – математических наук Хайдаров А.

старший научный сотрудник
Института Математики и
Информационных технологии
кандидат физика – математических наук Нуралиев Ф.Н.

1. Языки и грамматики

Алфавит - это любое множество символов. Понятие символа не определяется. Цепочка символов 0,1,2 записывается как "012" (или 012). Другие обозначения:

xR - цепочка x с символами в обратном порядке
x^n - цепочка x, повторенная n раз
x* - цепочка x, повторенная 0 или более раз
x+ - цепочка x, повторенная 1 или более раз
xy - сцепление (конкатенация) цепочек x и y
|x| - длина (число символов) цепочки x
e - пустая цепочка

Цепочку из одного символа будем обозначать самим символом. Буквы x,y,z,u,v,w,t будем применять для обозначения цепочек. Множество всех цепочек из элементов множества E естественно обозначить через E*. Язык - это подмножество E*. Примеры языков: Си, русский, { 0 1 | n >= 0 }.

Язык можно задать:

- перечислив все цепочки;
- написав программу-распознаватель, которая получает на вход цепочку символов и выдает ответ "да", если цепочка принадлежит языку и "нет" в противном случае;
- с помощью механизма порождения - грамматики.

Чтобы задать грамматику, требуется указать:

- множество символов алфавита (или терминальных символов) E. Будем обозначать их строчными символами алфавита и цифрами;
- множество нетерминальных символов (или метасимволов), не пересекающееся с E со специально выделенным начальным символом S. Будем обозначать их прописными буквами;
- множество правил вывода, определяющих правила подстановки для цепочек. Каждое правило состоит из двух цепочек (например, x и y), причем x должна содержать по крайней мере один нетерминал; и означает, что цепочку x в процессе вывода можно заменить на y. Вывод цепочек языка начинается с нетерминала S. Правило грамматики будем записывать в виде x : y.

(Также употребляется запись x ::= y или x -> y)

Более строго, определим понятие выводимой цепочки:

- S - выводимая цепочка;
- если xyz - выводимая цепочка и в грамматике имеется правило y:t, то xtz - выводимая цепочка;
- определяемый грамматикой язык состоит из выводимых цепочек, содержащих только терминальные символы.

Примеры:

а) S : e
S : 0S1

б) S : e
S : (S)
S : SS

Для сокращения записи принято использовать символ "или" - "|".
Короткая форма записи предыдущих примеров:

а) S : e | 0S1 б) S : e | (S) | SS

Форма Бэкуса-Наура

Грамматики в свою очередь образуют т.н. метаязык. Выше была описана "академическая" форма записи метаязыка. На практике применяется также другая форма записи, традиционно называемая нормальными формами Бэкуса-Наура (НФБН). Терминалы в НФБН записываются как обычные символы алфавита, а нетерминалы - как имена в угловых скобках <>. Например, грамматику целых чисел без знака можно записать в виде:

<число> : <цифра> | <цифра> <число>
<цифра> : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Рассмотрим язык простейших арифметических формул:
<формула> : (<формула>) | <число> | <формула><знак><формула>
<знак> : + | *
Почему "3+5*2" является формулой? Приведем последовательность преобразований цепочек (так называемый "разбор" или "вывод"):

<формула> : <формула> <знак><формула> :
<формула><знак><формула><знак><формула> :
<число> <знак><формула><знак><формула> :
3 <знак><формула><знак><формула> :
3 + <формула><знак><формула> :
3 + <число> <знак><формула> :
3 + 5 <знак><формула> :
3 + 5 * <формула> :
3 + 5 * <число> :
3 + 5 * 2

Сокращенно наличие вывода (цепочки преобразований) будем записывать в виде <формула>:3+5*2. Большинство грамматик допускают несколько различных выводов для одной и той же цепочки из языка. Если в процессе вывода цепочки правила грамматики применяются только к самому левому нетерминалу, говорят, что получен левый вывод цепочки. Аналогично определяется правый вывод.

Дерево вывода имеет ветви (линии) и узлы (помечены терминалами и нетерминалами), из которых растут ветви. Конечные узлы (терминалы) называются листьями. Понятия "поддерево", "корень дерева", видимо, не нуждаются в определении.

Одно и то же дерево разбора может описывать различные выводы (в дереве не фиксирован порядок применения правил). Однако, между

левыми (или правыми) выводами и деревьями разбора для цепочек существует однозначное соответствие (упражнение).

Если для одной и той же цепочки можно изобразить два разных дерева разбора (или, что то же самое, построить, два разных правых вывода), грамматика называется неоднозначной. Описанная грамматика неоднозначна. Тот же самый язык можно описать однозначной грамматикой:

<формула> : <терм> | <терм><знак><формула>
<терм> : (<формула>) | <число>
<знак> : + | *

Дерево разбора определяет некоторую структуру цепочки языка. Так, мы видим, что подцепочка "3+5" является "формулой". Это противоречит нашим (интуитивным) понятиям о смысле исходной формулы: 3+5 в отличие от 5*2 не является подвыражением. Мы можем учесть приоритет операций, изменив грамматику:

<формула> : <терм> | <формула> + <терм>
<терм> : <элемент> | <терм> * <элемент>
<элемент> : (<формула>) | <число>

Если в левой и правой частях правила начинаются с одного нетерминала говорят, что грамматика леворекурсивна.
Устранение левой рекурсии:

<формула> : <терм> | <терм><плюсминус><формула>
<терм> : <элемент> | <элемент><умножитьразделить><терм>
<плюсминус> : + | -
<умножитьразделить> : * | /

Фактически, преобразовав грамматику, мы изменили порядок свертки операций. Традиционно операции одного приоритета выполняются слева направо (говорят, что операции левоассоциативны), а только что написанная грамматика определяет операции как правоассоциативные.

Использование метасимволов

В правилах грамматики могут встречаться специальные символы-метасимволы.

В качестве метасимволов чаще всего используются следующие символы:
() (круглые кавычки), [] (квадратные скобки), {} (фигурные скобки), “”(кавычки), (запятая).

Круглые кавычки означают, что из перечисленных внутри них цепочек в данном месте может стоять только одна цепочка.

Квадратные кавычки означают, что указанная в них внутри них цепочка в данном месте может встречаться, а может и не встречаться..

Фигурные кавычки означают, что указанная в них внутри них цепочка в данном месте может не встречаться, а может встречаться один или сколько угодно много раз.

Запятая служит для разделения цепочек внутри круглых скобок.

Точка служит для разделения цепочек внутри круглых скобок.

Кавычки служат для того, чтобы один из метасимволов включить обычным образом.

С применением метасимволов грамматика арифметических формул легко запишется без левой рекурсии:

<формула> : <терм> { <плюс минус> <формула> }

<терм> : <элемент> { <умножить разделить> <элемент> }

Упражнения

Определить грамматику в обычной форме и форме Бэкуса-Наура. Построит вывод конкретной цепочки на основе грамматики.

1. Паскаль – грамматика выражений для арифметических операций, условных выражений, присваивания и скобок.
2. Паскаль – грамматика выражений для сравнения, логических операций, присваивания и скобок.
3. Паскаль – грамматика для определения с помощью **type** новых типов данных и определения, переменных на их основе.
4. Паскаль – грамматика определений простых переменных и указателей.
5. Паскаль – грамматика определения массивов и массивов указателей.
6. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;** **if** (с else и без него), и блочных скобок.
7. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;, case** и блочных скобок.
8. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;, for,** и блочных скобок.
9. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;, while** и блочных скобок.
10. Паскаль – грамматика арифметических выражений со скобками, операторов **<выражение>;** и **<ид>=<выражение>;, repeat-until** и блочных скобок.
11. Паскаль – грамматика структурированного типа **record** и контекстное определение, переменных на его основе.
12. Паскаль – грамматика структурированного типа **record** и контекстное определение, переменных и массивов на его основе.
13. Паскаль – грамматика определения функции вида **i(,i,i,i...){ E=i;E=i;E; }** Программа представляет собой последовательность определений функций. Выражение может содержать вызов функции вида **i(E,E,E,...).**
14. Паскаль – грамматика контекстного определения переменных и определения функций.
15. Паскаль – грамматика контекстного определения переменных и определения процедур.

2. Формальные грамматики

Понятие формальной грамматики

Формальная грамматика - это четверка $G = <V_N, V_T, P, S>$, в которой

V_N - ***нетерминальный словарь*** (множество нетерминальных символов);

V_T - ***терминальный словарь*** (множество терминальных символов);

P - ***множество грамматических правил***;

$S \in V_N$ - ***начальный нетерминальный символ***.

Метаязык - язык, с помощью которого описывается язык:

:= - есть по определению;

| - "исключающее или";

< ... > - внутри – один нетерминальный символ;

[ ] - необязательная часть;

, - запятая – разделитель при перечислении.

Пример: Построим грамматику G_1:

<прог>:=<оп> | <оп>;<прог>

<оп>:=<пер> := <выр>

<пер>:=a | b | c

<выр>:=<пер> | <пер> <зн> <выр>

<зн>:= + | - | * | /

$V = V_N \cup V_T$ - ***обобщенный словарь***.

V^* - *цепочка символов (строка, слово) из обобщенного слова*

V^*_N - *цепочка символов (строка, слово) из нетерминального словаря;*

V^*_T - *цепочка символов (строка, слово) из терминального словаря.*

$\varepsilon \in V$ - *пустой символ*, входит в обобщенный словарь.

Строка α *непосредственно порождает* строку β и обозначается: $\alpha \Rightarrow \beta$, если $\alpha = vxw$ $\beta = vyw$ и существует некоторое правило $p: x := y$, где $v, w, \in V^*, x \in V_N, y = V^* \setminus \{\varepsilon\}$

Строка α *порождает* строку β и обозначается $\alpha \Rightarrow^* \beta$, когда от строки α можно перейти к строке β с помощью последовательности непосредственных порождений.

Продолжая пример:

<прог> $\Rightarrow$ <оп>; <прог> $\Rightarrow$ <оп>; <оп> $\Rightarrow$ <пер> := <выр>; <оп> $\Rightarrow^*$ a := b + c; c := a + b - c;

Грамматика, использующая процедуры (непосредственного) порождения – порождающая грамматика.

Строка β *непосредственно свертывается* в строку α и обозначается: $\alpha \Leftarrow \beta$, если $\alpha = vxw$ $\beta = vyw$ и существует некоторое правило $p: x := y$, где $v, w, \in V^*, x \in V_N, y = V^* \setminus \{\varepsilon\}$

Строка β *свертывается* в строку α и обозначается $\alpha \ ^*\Leftarrow \beta$, когда от строки β можно перейти к строке α с помощью последовательности непосредственных свертываний.

Грамматика, использующая процедуры (непосредственного) свертывания – распознающая грамматика.

Строки символов из обобщенного словаря, получающиеся в процессе порождения или свертывания, называются **сентенциальными формами.**

Язык L, порождаемый данной грамматикой G - множество нетерминальных цепочек, порождаемых из начального нетерминального символа. Такие терминальные цепочки называются **предложениями** данного языка.

$$L(G) = \{ x \in V^*_N \mid S \Rightarrow^* x \}$$

Аналогично можно определить язык L через свертывание.

$$L(G) = \{ x \in V^*_N \mid S \Leftarrow^* x \}$$

Замечание. Другой вариант метаязыка

вместо ::= используется стрелка → , терминальные символы записываются маленькими (строчными) буквами, а нетерминальные – большими (прописными) буквами.

Такой вариант мета языка чаще используется в математической литературе. Первый предпочитают использовать в литературе для программистов.

Классификация языков по Хомскому

Н. Хомский предложил подразделять формальные грамматики, как и порождаемые ими языки на четыре типа.

Тип 0 - формальная грамматика, на правила которой не накладывается никаких ограничений. Для исследования они интереса не представляют – поскольку режим «делай, что хочешь» для математики и для практики редко представляют интерес.

Тип 1. К этому типу относятся грамматики, правила которых имеют вид:

$$v\alpha w ::= v\beta w,$$

где

$v, w \in V^*$ - произвольные цепочки символов, возможно пустые;

$\alpha \in V_N$ - нетерминальный символ;

$\beta \in V^* \setminus \{\varepsilon\}$ [иногда $\beta \in V^*$].

[$\beta \in V^*$].

Эти грамматики еще называют ***контекстно-зависимыми*** или ***КЗ-грамматиками.***

Здесь строка α заменяется на строку β в рамках какого-то контекста. Часто используется на практике подмножество таких грамматик, называемое *грамматиками непосредственных составляющих:*

$$vAw ::= v\alpha w, \text{ где } v, w, \alpha \in V^*, \ A \in V_N$$

При этом часто рассматриваются *неукорачивающиеся* грамматики (что обеспечивается непустой цепочкой β).

Тип 2. К этому типу относятся грамматики, правила которых имеют вид:

$\alpha ::= \beta$

$\alpha \in V_N$ - нетерминальный символ;

$\beta \in V^* \setminus \{\varepsilon\}$:

Здесь также представляют наибольший интерес грамматики непосредственных составляющих.

Такие грамматики называются ***контекстно-свободными*** грамматиками или ***КС-грамматиками.***

Языки программирования большей частью описываются грамматиками этого типа.

Тип 3. К этому типу относятся грамматики, правила которых имеют один из двух видов:

$$\begin{pmatrix} A := Bc \\ A := b \end{pmatrix}$$

где $A, B, C \in V_N$; $b, c \in V_T$;

Это так называемый *леворекурсивный вариант.* В качестве альтернативы возможен и праворекурсивный вариант:

$$\begin{pmatrix} A := cB \\ A := b \end{pmatrix}$$

Такие грамматики называют ***регулярными*** или ***автоматными грамматиками.*** Лексический анализ в компиляторе обычно наиболее целесообразно описывать с помощью этих грамматик.

Пример 1. Пусть имеется G - грамматика с правилами вывода

I -> CD Пример вывода

C -> aCA	I -> CD
C -> bCB	-> aCAD
AD -> aD	-> abCBAD
BD -> bD	-> abBAD
Aa -> aA	-> abBaD
Ab -> bA	-> abaBD
Ba -> aB	-> ababD
Bb -> bB	-> abab
C -> e	
D -> e	

Грамматика без ограничений на правила вывода, тип 0.

Пример 2. Пусть G определяется правилами

I -> aIBC Пример вывода

I -> abC	I -> aIBC
CB -> BC	-> aabCBC
bB -> bb	-> aabBCC
bC -> bc	-> aabbCC
cC -> cc	-> aabbcC
	-> aabbcc

Эта грамматика порождает язык $\{a^n b^n c^n \mid n>=1\}$ и является контекстно-зависимой (неукорачивающей).

Пример 3. Пусть G - грамматика с V={a,+,*,(,)}, W={I,A,B} и правилами
I -> I+A Пример вывода

I -> A	I -> I+A
A -> A*B	-> A+A
A -> B	-> B+A
B -> (I)	-> a+A
B -> a	-> a+A*B
	-> a+B*B
	-> a+a*B
	-> a+a*a

Язык представляет собой множество арифметических выражений.
Контекстно-свободная грамматика.
Пример 4. Множество нечетных чисел в унарном представлении - это множество цепочек вида a, aaa, aaaaa,..., т.е. язык {a }. Он порождается грамматикой с правилами I -> a; I -> aaI.

Упражнения

1. Определить тип грамматики. Описать язык, порождаемый этой грамматикой.

I->bII | a

2. Определить тип грамматики. Описать язык, порождаемый этой грамматикой.

I -> aIb I -> ab

3. Определить тип грамматики. Описать язык, порождаемый этой грамматикой.

$S \rightarrow 0A1 \mid 01$
$0A \rightarrow 00A1$
$A \rightarrow 01$

4. Определить тип грамматики. Описать язык, порождаемый этой грамматикой.

$S \rightarrow 0A1$
$0A \rightarrow 00A1$
$A \rightarrow \varepsilon$

5. Определить тип грамматики. Описать язык, порождаемый этой грамматикой. Написать для этого языка КС-грамматику.

$S \rightarrow aAb$
$aA \rightarrow aaAb$
$A \rightarrow \varepsilon$

6. Определить тип грамматики. Описать язык, порождаемый этой грамматикой. Написать для этого языка КС-грамматику.

$S \rightarrow AB \mid ABS$
$AB \rightarrow BA$

$BA \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

7. Построить регулярную грамматику, порождающую цепочки в алфавите {a,b}, в которых символ a не встречается два раза подряд.

8. Построить регулярную грамматику, эквивалентную грамматике с правилами:

$S \rightarrow A\,A$
$A \rightarrow B \mid BA$
$B \rightarrow 0 \mid 1$

9. Построить регулярную грамматику, эквивалентную грамматике с правилами:

$S \rightarrow A \mid AS$
$A \rightarrow a \mid bb$

10. Написать КС-грамматику для языка L, и левосторонний вывод для цепочки aabbbcccc. $L = \{a^{2n}b^{m}c^{2k} \mid m=n+k, m>1\}$.

11. Построить грамматику, порождающую сбалансированные относительно круглых скобок цепочки в алфавите { a, (,), ⊥}. Сбалансированную цепочку α определим рекуррентно: цепочка α сбалансирована, если

а) α не содержит скобок,

b) $\alpha = (\alpha_1)$ или $\alpha = \alpha_1\alpha_2$, где α_1 и α_2 сбалансированы.

12. Построить грамматику, порождающую язык:

$L = \{a^{n}b^{m} \mid n, m \geq 1\}$

13. Построить грамматику, порождающую язык L, и вывод для цепочки aacbbbcaa в этой грамматике.

$L = \{a^{n}cb^{m}ca^{n} \mid n, m>0\}$

14. Дан язык $L = \{1^{3n+2}0^{n} \mid n \geq 0\}$. Определить его тип, написать грамматику, порождающую L. Построить левосторонний и правосторонний выводы для цепочки 1111111100.

15. Построить грамматику, порождающую язык L, и вывод для цепочки 110000111 в этой грамматике.

$L = \{1^{n}0^{m}1^{p} \mid n+p>m; n, p, m>0\}$

3. Регулярные языки и грамматики

Распознающие автоматы

Недетерминированный конечный автомат (НКА) - это пятерка M=(Q,T,D,Q0,F), где

(1) Q - конечное множество состояний;

(2) T - конечное множество допустимых входных символов;

(3) D - функция переходов, отображающая множество QxT во множество подмножеств множества Q и определяющая поведение управляющего устройства;

(4) Q0<=Q - множество начальных состояний управляющего устройства;

(5) F<=Q - множество заключительных состояний.

Детерминированный конечный автомат (ДКА) - это пятерка M=(Q,T,D,q0,F), где

(1) Q - конечное множество состояний;

(2) T - конечное множество допустимых входных символов;

(3) D - функция переходов, отображающая множества QxT в множество Q и определяющая поведение управляющего устройства;

(4) q0<-Q - начальное состояние управляющего устройства;

(5) F<=Q - множество заключительных состояний.

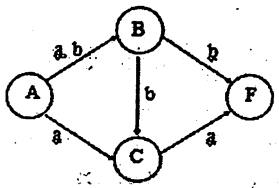
Работа конечного автомата представляет собой некоторую последовательность шагов, или тактов. Такт определяется текущим состоянием управляющего устройства и входным символом, обозреваемым в данный момент входной головкой. Сам шаг состоит из изменения состояния и сдвига входной головки на одну ячейку вправо.

Текущее состояние управляющего устройства, символ под головкой и цепочка символов вправо от головки называются конфигурацией автомата. Конфигурация (q0,w) называется начальной, а пара (q,e), где q<-F, называется заключительной (или допускающей). Такт автомата M представляется бинарным отношением |-, определенным на конфигурациях: отношение имеет место, если есть переход из конфигурации (q1,w1) в конфигурацию (q2,w2). Отношения |-+ и |-* - это, соответственно, транзитивное и рефлексивно-транзитивное замыкание отношения |-. Говорят, что автомат M допускает цепочку w, если (q0,w)|-*(q,e) для некоторого q<-F. Языком, допускаемым (распознаваемым, определяемым) автоматом M, (обозначается L(M)), называется множество входных цепочек, допускаемых автоматом M. Т.е.

L(M)={w | w<-T* и (q0,w)|-*(q,e) для некоторого q<-F}

Конечный автомат может быть изображен графически в виде графа, в котором каждому состоянию соответствует вершина, а дуга, помеченная символом a, соединяет две вершины p и q, если функция переходов содержит (q,a)->p. На диаграмме выделяются конечные состояния (в примерах выше двойным контуром).

Распознающий автомат – это, как правило, *недетерминированный частичный автомат*. То есть по одному и тому же сигналу можно перейти в различные состояния, а в некоторых состояниях нет перехода для ряда входных сигналов.



	0	0	0	1
	A	B	C	F
a	B,C		F	
b	B	C, F		

Кстати, строка, приписывающая состояниям выходные сигналы совсем не обязательна.

Представление этого автомата с помощью автоматной грамматики:

$A \to aB \mid bB \mid aC$

$B \to bC \mid b$

$C \to a$

Это праворекурсивная автоматная грамматика.

Переход от недетерминированного распознающего автомата к детерминированному

Состояния автомата и совокупности состояний, в который автомат переходит, объявляются множествами. Каждое из этих множеств становится состоянием нового детерминированного автомата. Переход из состояния, содержащего множество элементов, будет в состояние-множество, составленное из всех состояний, в которые в исходном автомате осуществлялись переходы. Заметим, что пустые клеточки дают состояние - пустое множество.

$A \to aB \mid bB \mid aC$

$B \to bC \mid b$

$C \to a$

	A	B	C	F
a	B,C		F	
b	B	C,F		

	{A}	{B,C}	{B}	{F}	{C,F}	{}
a	{B,C}	{F}	{}	{}	{F}	{}
b	{B}	{C,F}	{C,F}	{}	{}	{}

Переход от праволинейной грамматики к автоматной

Праволинейная грамматика - грамматика с правилами вида:

$A \to \alpha$

$A \to \alpha B$

где $A, B \in V_N, \alpha \in V_T^*$

То есть это такая КС-грамматика, где вначале идет любое количество терминальных символов, а в конце возможен один нетерминальный символ

Пример:

Дана праволинейная грамматика:

1. S→aA
2. S→bc
3. S→A
4. A→abbS
5. A→cA
6. A→E

Правила 2,3,4 – нарушают требования к автоматным грамматикам.
Их можно последовательно заменить совокупностями автоматных правил.

Регулярные множества и регулярные выражения

Пусть T - конечный алфавит. Регулярное множество в алфавите T определяется рекурсивно следующим образом (знаком '<-' будем обозначать принадлежность множеству, знаком '<=' включение):

(1) {} (пустое множество) - регулярное множество в алфавите T;

(2) {a} - регулярное множество в алфавите T для каждого a<-T;

(3) {e} - регулярное множество в алфавите T (e - пустая цепочка);

(4) если P и Q - регулярные множества в алфавите T, то таковы же и множества

(а) P U Q (объединение),

(б) PQ (конкатенация, т.е. множество pq, p<-P, q<-Q),

(в) P* (итерация: P*={e} U P U PP U...;

(5) ничто другое не является регулярным множеством в алфавите T.

Итак, множество в алфавите T регулярно тогда и только тогда, когда оно либо {}, либо {e}, ибо {a} для некоторого a<-T, либо его можно получить из этих множеств применением конечного числа операций объединения, конкатенации и итерации. Приведенное выше определение регулярного множества, одновременно определяет и форму его записи, которую будем называть регулярным выражением. Для сокращенного обозначения выражения PP* будем пользоваться записью P+ и там, где это необходимо, будем использовать скобки. В этой записи наивысшим приоритетом обладает операция *, затем конкатенация и, наконец, операция U, для записи которой иногда будем использовать значок '|'. Так, 0|10* означает (0|(1(0*))).

Кроме того, мы будем использовать запись вида

d1 = r1
d2 = r2
.....
dn = rn

где di - различные имена, а каждое ri - регулярное выражение над символами T U {d1,d2,...,di-1}, т.е. символами основного алфавита и ранее

определенными символами. Таким образом, для любого Гі можно построить регулярное выражение над Т, повторно заменяя имена регулярных выражений на обозначаемые ими регулярные выражения.

Несколько примеров регулярных выражений и обозначаемых ими множеств

Идентификатор - это регулярное выражение

Идентификатор = Буква (Буква|Цифра)*

Буква = {a,b,...,z}

Цифра = {0,1,...,9}

Число в десятичной записи - это регулярное выражение

Целое = Цифра+

Дробная_часть = .Целое | e

Степень = (E (+ | - | e) Целое) | e

Число = Целое Дробная_часть Степень

Ясно, что для каждого регулярного множества можно найти по крайней мере одно регулярное выражение, обозначающее это множество. И обратно: для каждого регулярного выражения можно построить регулярное множество, обозначаемое этим выражением. Для каждого регулярного множества существует бесконечно много обозначающих его регулярных выражений. Будем говорить, что два регулярных выражения равны, если они обозначают одно и то же множество.

Задачи и упражнения

1. Написать регулярную грамматику и регулярное выражение, порождающие тот же язык, что и грамматика:

S : A B ; A : X|Y ; X : x|xX ; Y : y|yY ; B : b|bB

2. Построить конечный автомат и регулярное выражение для данной регулярной грамматики:

S → A ⊥

A → Ab | Bb | b

B → A a

3. Построить регулярную грамматику и конечный автомат, соответствующие регулярному выражению:

(101)*(010)*

4. Постройте недетерминированный, а затем детерминированный конечные автоматы, воспринимающие регулярное выражение:

(101)*(110)*

5. Написать регулярное выражение, описывающее следующий язык - слово, буквы которого расположены в алфавитном порядке

- последовательность цифр, в которой нет повторяющихся цифр
- последовательность цифр, в которой не более чем одна цифра встречается несколько раз
- строка из нулей и единиц, в которой нет подстроки 0 0 1
- строка из нулей и единиц, в которой нет подпоследовательности 0 0 1

6. Напишите регулярной грамматикой и приведите регулярное выражение для идентификатора Фортрана. (До шести букв или цифр, первый символ должен быть буквой).

7. Постройте конечный автомат, который будет распознавать слова GO TO.

8. Постройте конечные распознаватели для описанных ниже цепочек из нулей и единиц. Затем превратите каждый распознаватель в процессор с концевым маркером.

а) Число единиц четное, а число нулей - нечетное.

б) Каждый третий символ единица.

в) За каждым вхождением пары 11 следует 0.

9. Постройте конечный автомат с входным алфавитом {0,1},который допускает такое множество цепочек:

а) Входную цепочку 101.

б) Все входные цепочки, кончающиеся на 1 и начинающиеся с 0.

в) Все цепочки, содержащие три единицы.

г) Все цепочки, в которых перед и после каждой единицы стоит 0.

10. Постройте конечный автомат, который будет распознавать слова STEP и STRING.

11. Построить недетерминированный автомат, который допускает только две цепочки АЛГОЛ и ПАСКАЛЬ.

12. Постройте конечный автомат для распознавания вещественных констант. Он должна воспринимать константы вида:

1 2 -0.4 +3.14 .2 1. -1.e38 1e+2 +0.5e-23 и т.д.

13. Построить минимальные детерминированные конечные автоматы для следующих регулярных выражений

(a|b)*a(a|b)

(a|b)*a(a|b)(a|b)

(a|b)*a(a|b)(a|b)(a|b)

Доказать, что любой детерминированный конечный автомат для выражения (a|b)a(a|b)(a|b)...(a|b), содержащего n-1 (a|b) в конце содержит не менее чем 2**n состояний.

14. Построить конечный автомат и регулярное выражение, которое проверяет, является ли имя файла частным случаем разновидности регулярного выражения, в котором метасимвол * обозначает любую (в том числе и пустую) последовательность символов кроме ".", а метасимвол ? - любой символ кроме "."

15. Постройте конечный автомат, который распознает химические формулы, составленные из элементов H,C,N,O,S. Элементы в формулах могут появляться в любом порядке и в любых сочетаниях.

4. КС - Грамматики

Основные определения

Пусть G= - контекстно-свободная грамматика, где N - множество нетерминальных символов, T - множество терминальных символов, P -

множество правил вывода и S - аксиома. Будем говорить, что uxv выводится за один шаг из uAv (и записывать это как uAv=>uxv), если A->x - правило вывода и u и v - произвольные строки из (N U T)*. Если u1=>u2=>...=>un, будем говорить, что из u1 выводится un, и записывать это как u1=>*un.

Т.е.:

1) u=>*u для любой строки u,

2) если u=>*v и v=>*w, то u=>*w.

Аналогично, "=>+" означает выводится за один или более шагов. Если дана грамматика G с начальным символом S, отношение =>+ можно использовать для определения L(G) - языка, порожденного G. Строки L(G) могут содержать только терминальные символы G. Строка терминалов w принадлежит L(G) тогда и только тогда, когда S=>+w. Строка w называется предложением в G. Если S=>*u, где u может содержать нетерминалы, то u называется сентенциальной формой в G. Предложение - это сентенциальная форма, не содержащая нетерминалов.

Рассмотрим выводы, в которых в любой сентенциальной форме на каждом шаге делается подстановка самого левого нетерминала. Такой вывод называется левосторонним. Если S=>*u в процессе левостороннего вывода, то u - левая сентенциальная форма.

Аналогично определяется правосторонний вывод.

Упорядоченным графом называется пара (V,E), где V обозначает множество вершин, а E - множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид ((v,e1),(v,e2),...,(v,en)). Этот элемент указывает, что из вершины v выходят n дуг, причем первой из них считается дуга, входящая в вершину e1, второй - дуга, входящая в вершину e2, и т.д.

Дерево вывода в грамматике G=(N,T,P,S) - это помеченное упорядоченное дерево, каждая вершина которого помечена символом из множества N U T U {e}. Если внутренняя вершина помечена символом A, а ее прямые потомки - символами X1,..., Xn, то A->X1X2...Xn - правило этой грамматики. Упорядоченное помеченное дерево D называется деревом вывода (или деревом разбора) в КС-грамматике G(S)=(N,T,P,S), если выполнены следующие условия:

(1) корень дерева D помечен S;

(2) каждый лист помечен либо терминальным символом, либо e;

(3) каждая внутренняя вершина помечена нетерминалом;

(4) если N - нетерминал, которым помечена внутренняя вершина и X1,...,Xn - метки ее прямых потомков в указанном порядке, то N->X1...Xk - правило из множества P.

Пример деревьев вывода

Пусть дана грамматика.

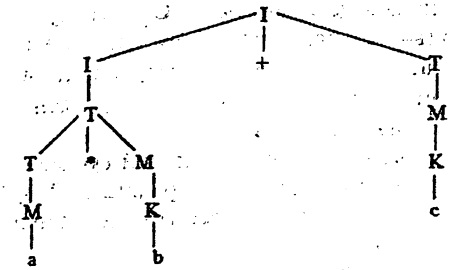
$I \rightarrow T$

$I \rightarrow I + T$

I → I - T
T → M
T → T*M
T → T/M
M → (I)
M → K
K → a
K → b
K → c

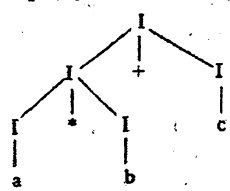
Построим дерево вывода.

Для предложения a * b + c дерево вывода будет:



Этот же результат можно получить и другим способом:

I → I + I
I → I - I
I → I*I
I → I/I
I → (I)
I→a
I → b
I → c



Левый, левосторонний вывод - когда заменяется самый левый нетерминальный символ. Правый, правосторонний вывод - когда заменяется самый правый нетерминальный символ.

Когда одна цепочка может иметь несколько деревьев вывода, то соответствующая грамматика неоднозначна.

Приведение КС-грамматик

Для любого КС-языка существует бесконечное число грамматик, порождающих этот язык. Поэтому возникает проблема выбора грамматики, наиболее подходящей по тем или иным свойствам.

Приводятся некоторые эквивалентные преобразования для улучшения грамматик.

Нетерминальный символ называется существенным, если он является достижимым (I=> aAb) и продуктивным, производящим (A=> g), в противном случае он называется несущественным или бесполезным.

Грамматика называется приведенной, если она неукорачивающая и не содержит бесполезных символов.

Для любой КС-грамматики можно построить эквивалентную ей приведенную КС-грамматику. Приведенная грамматика G'=(V',W',P',I) эквивалентная исходной, определяется так: W'- множество существенных нетерминальных символов, P'- только те правила из P, в которых нет бесполезных символов, V'- терминальные символы, которые встречаются в правых частях правил из P.

Из эквивалентности грамматик следует равенство L(G)=L(G').

Процедура 1 обнаружения бесплодных нетерминалов.

Нетерминальный символ является продуктивным, если из него выводится какая-нибудь терминальная цепочка, иначе он является бесплодным.

Шаг 1. Составить список нетерминалов в правилах, правая часть которых не содержит нетерминальных символов.

Шаг 2. Если правая часть правила содержит нетерминалы, занесенные в список, то занести в него и нетерминалы левой части.

Полученный список содержит только продуктивные нетерминалы.

Пример. Рассмотрим грамматику

1. I -> aABI
2. I -> bcACd I -> bcACd
3. A -> bAB
4. A -> cIA A -> cIA
5. A -> aCC A -> aCC
6. B -> bAB
7. B -> cIB
8. C -> cI C -> cI
9. C -> c C -> c

На шаге 1 в список можно занести нетерминал C из правила 9.

На шаге 2 в список можно добавить нетерминал A из правила 5, а затем нетерминал I из правила 2.

Продуктивными нетерминалами являются C,A,I, а оставшийся нетерминал B - бесплодный. Удалив все правила, связанные с B, получим эквивалентную грамматику.

Процедура 2 обнаружения недостижимых нетерминалов. Если нетерминал в левой части правила является достижимым, то достижимы и все символы правой части правила.

Шаг 1. Образовать одноэлементный список из начального нетерминала.

Шаг 2. Если левая часть правила содержит нетерминалы, включенные в список, то включить в него и нетерминалы правой части.

Полученный список содержит только достижимые нетерминалы.

Пример. Рассмотрим грамматику

1.	I -> aAB	I -> aAB
2.	I -> E	I -> E
3.	A -> dDA	A -> dDA
4.	A -> e	A -> e
5.	B -> bE	B -> bE
6.	B -> f	B -> f
7.	C -> cAB	
8.	C -> dID	
9.	C -> a	
10.	D -> eA	D -> eA
11.	E -> fA	E -> fA
12.	E -> g	E -> g

На шаге 1 в список заносится символ I.

На шаге 2 добавляются в список нетерминалы A,B,E из правил 1 и 2. Затем нетерминал D из правила 3.

Достижимыми являются нетерминалы I,A,B,D,E, а нетерминал C недостижимым. Удалив правила с нетерминалом C, получим эквивалентную грамматику.

Пример. Проиллюстрировать применение обеих процедур на примере грамматики G

Исходная	После 1 процедуры	После 2 процедуры
I -> ac	I -> ac	I -> ac
I -> bA	C -> bC	
A -> cBC	C -> d	
B -> aIA	A и B - бесплодны	C - недостижим
C -> bC		
C -> d		

Следующее полезное преобразование грамматик - это устранение правил вида A -> B, которые называются цепными.

Процедура 3 исключения цепных правил.

Шаг 1. Для каждого A(W находится множество W всех нетерминальных символов E, таких, что A => E (транзитивное замыкание отношения A => E если имеется последовательность правил A ->B, B->C,... -> E).

Шаг 2. Определяется множество правил P': если B->a содержится в P и не является цепным, то для любого A такого, что B(W , включаем в P' правило A->a.

При :р. Рассмотрим грамматику

1.	I -> T	I -> I+T	T -> T*M	M -> (I)
2.	I -> I+T	I -> I-T	T -> T/M	M -> a
3.	I -> I-T	I -> T*M	T -> (I)	M -> b
4.	T -> M	I -> T/M	T -> a	M -> c
5.	T -> T*M	I -> (I)	T -> b	
6.	T -> T/M	I -> a	T -> c	
7.	M -> (I)	I -> b		
8.	M -> K	I -> c		
9.	K -> a			
10.	K -> b			
11.	K -> c			

Упражнения

1. Найдите КС-грамматику для каждого из следующих языков
   а) {1 0 |n>m>0};
   б) {1 0 1 0 |n,m>=0};
   в) {1 0 1 0 |n,m>=0};
   г) все цепочки в алфавите {0,1} содержащие равное количество нулей и единиц

2.Опишите языки порождаемые следующими грамматиками

а)	<I> -> 10<I>0	<I> -> a<a>
	<a> -> b<a>	<a> -> a
б)	<I> -> <I><I>	<I> -> 1a>0
	<a> -> 1<a>0	<a> -> e
в)	<I> -> 1<a>	<I> -> <B>0
	<a> -> 1<a>	<a> -> <C>
	<B> -> <B>0	<B> -> <C>
	<C> -> 1<C>0	<C> -> e
г)	<I> -> a<I><I>	<I> -> a

3. Построить дерево с левосторонним выводом в грамматике
   1. <I> -> a<a><B>c
   2. <I> -> e
   3. <a> -> c<I><B>
   4. <a> -> <a>b
   5. <B> -> b<B>
   6. <B> -> a

4. Грамматика для арифметических выражений имеет вид
   1. <I> -> <I>+<T>
   2. <I> -> <T>
   3. <T> -> <T>*<F>
   4. <T> -> <F>
   5. <F> -> <F>I<P>
   6. <F> -> <P>

7. <P> -> (<I>)
8. <P> -> E

где E -представляет произвольное целое число. Привести дерево вывода в этой грамматике выражения 1+2*3+(5+6)*7

5. Какие из приведенных ниже цепочек можно вывести в данной грамматике? В каждом случае постройте левый вывод правый вывод и дерево вывода

<I> -> a<a>c<B>
<I> -> <B>d<I>
<B> -> a<I>c<a>
<B> -> c<a><B>
<a> -> <B>a<B>
<a> -> a<B>c
<a> -> a
<a> -> b

а) aacb
б) aacbccb
в) aababcbadcd

6. Сделайте вывод арифметического выражения
а) a*b+c
б) a+b*c
в) (a+b)*c
г) a*(b+c)
д) (a+b)/c
е) a/(b+c)

в грамматике
1. I -> T
2. I -> I+T
3. I -> I-T
4. T -> M
5. T -> T*M
6. T -> T/M
7. M -> (I)
8. M -> K
9. K -> a
10. K -> b
11. K -> c

7. Дана грамматика. Построить вывод заданной цепочки.

S → T | T+S | T-S
T → F | F*T
F → a | b b

Цепочка a-b∗a+b.

8. Дана грамматика. Построить вывод заданной цепочки.

S → aSBC | abC
CB → BC
B → bb
bC → bc
cC → cc

Цепочка aaabbbccc

9. Дана грамматика с правилами:

$S \rightarrow S0 \mid S1 \mid D0 \mid D1$

$D \rightarrow H.$

$H \rightarrow 0 \mid 1 \mid H0 \mid H1$

Построить восходящим и нисходящим методами дерево вывода для цепочки:
10.1001

10. Дана грамматика с правилами:

$S \rightarrow$ if B then S | B = E

$E \rightarrow B \mid B + E$

$B \rightarrow a \mid b$

Построить восходящим и нисходящим методами дерево вывода для цепочки:
if a then b = a+b+b

11. Найти приведенную грамматику для

I -> aABC I -> bCEI
I -> aE A -> bE
A -> ICD A -> d
B -> dFI B -> aBC
C -> aEI C -> bE
D -> aAC D -> d
E -> aCE E -> e
F -> AB F -> aF

12. Удалите бесплодные нетерминалы в грамматике

I -> aAI
I -> bAC
A -> bA
A -> cI
A -> cC
B -> bAB
B -> cI
C -> cIA

13. Удалите лишние нетерминалы в грамматике

I -> ac
I -> bA
A -> cBC
B -> aIA
C -> bC
C -> d

14. Приведите грамматику эквивалентную заданной

1. I -> T
2. I -> I+T
3. I -> I-T
4. T -> M
5. T -> T*M
6. T -> T/M
7. M -> (I)

8. M -> K
9. K -> a
10. K -> b
11. K -> c

15. Найти приведенную грамматику для

I -> aABC
I -> aE
A -> ICD
B -> dFI
C -> aEI
D -> aAC
E -> aCE
F -> A

5. LL грамматики

Методы и алгоритмы синтаксического анализа

Синтаксический разбор - построение дерева синтаксического разбора можно производить как сверху вниз - от начального нетерминала к предложению языка, так и снизу вверх - от предложения к начальному символу:

-нисходящий синтаксический разбор заключается в поиске замены очередного нетерминального символа в выводимой цепочке на правую часть соответствующего правила. При этом алгоритм может руководствоваться только "незакрытой", то есть нераспознанной частью предложения. Обычно для этого достаточно одного терминального символа.

-при восходящем синтаксическом разборе в предложении (или в промежуточной цепочке) ищется правая часть правила, которое необходимо "свернуть" к нетерминалу левой части. Решение принимается на основе анализа соседних терминальных символов в анализируемой цепочке.

По своей природе алгоритмы синтаксического разбора бывают детерминированные (сразу же строящие правильное дерево) и недетерминированные, то есть допускающие возврат на некоторое число шагов назад.

В процессе нисходящего разбора происходит замена каждого нетерминала, который встречается в выводимой цепочке на правую часть правила, в которой он находится в левой части. Очевидно, что самый простой алгоритм построения цепочки вывода заключается в полном переборе всех возможных цепочек. Тогда мы получаем классическую задачу поиска с полным перебором вариантов. Существуют типы грамматик, для которых данная задача может быть решена без перебора всех вариантов.

Множества FIRST и FOLLOW

При анализе входного слова полезными оказываются две функции, связанные с грамматикой G. Эти функции, FIRST и FOLLOW, позволяют построить таблицу, предсказывающего разбора для G, если, конечно, это возможно.

Множества, даваемые этими функциями, могут, кроме того, быть использованы при восстановлении после ошибок.

Если u - любая строка символов грамматики, положим FIRST(u) - множество терминалов, с которых начинаются строки, выводимые из u. Если u=>*e, то e также принадлежит FIRST(u).

Определим FOLLOW(A) для нетерминала A как множество терминалов a, которые могут появиться непосредственно справа от A в некоторой сентенциальной форме, т.е. множество терминалов a таких, что существует вывод вида S=>*uAav для некоторых u и v. Отметим, что между A и a в процессе вывода могут появиться нетерминальные символы, из которых выводится e. Если A может быть самым правым символом некоторой сентенциальной формы, то $ принадлежит FOLLOW(A). Здесь $ - символ окончания строки.

Для построения FIRST(X) для всех символов грамматики X применим следующий алгоритм.

Алгоритм. Построение множеств FIRST для символов грамматики.

Шаг 1. Если X - терминал, то FIRST(X) - это {X}; если X - нетерминал, полагаем FIRST(X)={}.

Шаг 2. Если имеется правило вывода X:e, то добавить e к FIRST(X).

Шаг 3. Пока ни к какому множеству FIRST(X) нельзя уже будет добавить новые элементы или e: если X - нетерминал и имеется правило вывода X: Y1Y2...Yk, то включить a в FIRST(X), если для некоторого i aєFIRST(Yi) и e принадлежит всем FIRST(Y1),...,FIRST(Yi-1), т.е. Y1...Yi-1=>*e. Если e принадлежит FIRST(Yj) для всех j=1,2,...,k, то добавить e к FIRST(X). Например, все, что принадлежит FIRST(Y1) принадлежит также и FIRST(X). Если из Y1 не выводится e, то ничего больше не добавляем к FIRST(X), но если Y1=>*e, то добавляем FIRST(Y2), и т.д.

Теперь FIRST для любой строки X1X2...Xn можно вычислить следующим образом.

Шаг 1. Полагаем FIRST(X1X2...Xn)={}.

Шаг 2. Добавим к FIRST(X1X2...Xn) все не e символы из FIRST(X1). Добавим также не e символы из FIRST(X2), если eєFIRST(X1),не e символы из FIRST(X3), если e принадлежит как FIRST(X1), так и FIRST(X2), и т.д.

Наконец, добавим e к FIRST(X1X2...Xn), если eєFIRST(Xi) для всех i.

Для вычисления FOLLOW(A) для нетерминала A применим следующий алгоритм.

Алгоритм. Построение FOLLOW(X) для всех X – нетерминалов грамматики.

Шаг 1. Положить FOLLOW(X)={}.
Шаг 2. Поместить $ в FOLLOW(S), где S - начальный символ и $ - правый концевой маркер.
Шаг 3. Если есть правило вывода A:uBv, то все из FIRST(v), за исключением e, добавить к FOLLOW(B).
Шаг 4. Пока ничего нельзя будет добавить ни к какому множеству FOLLOW(X): если есть правило вывода A:uB или A:uBv, где FIRST(v) содержит e (т.е. v=>*e), то все из FOLLOW(A) добавить к FOLLOW(B).

Таблицы предсказывающего анализатора

Таблицы предсказывающего анализатора - это таблица вида M[A,a], где A - нетерминал, и a - терминал.

Для конструирования таблицы анализа по грамматике G может быть использован следующий алгоритм.

Алгоритм. Построение таблиц предсказывающего анализатора.

Для каждого правила вывода A->u грамматики выполнить шаги 1 и 2

Шаг 1. Для каждого терминала a из FIRST(u) добавить A:u к M[A,a].

Шаг 2. Если e∈FIRST(u), добавить A:u к M[A,b] для каждого терминала b из FOLLOW(A). Если e∈FIRST(u) и $∈FOLLOW(A), добавить A:u к M[A,$].

Шаг 3. Положить все неопределенные входы равными error.

Пример. Рассмотрим грамматику арифметических выражений:

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid e$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid e$

$F \rightarrow (\ E\ ) \mid id$

Таблица анализа для нее изображена и приведена ниже. Здесь пустые клетки - входы ошибок. Непустые дают правила, по которым делается выбор нетерминала.

Нетерминал	Входной символ					
	id	+	*	(	)	$
E	E:TE'			E:TE'		
E'		E': +TE'			E' :e	E' :e
T	T:FT'			T:FT'		
T '		T':e	T ':*FT'		T':e	T ':e
F	F:id			F:(E)		

S и Q – грамматики

Рассмотрим грамматики, для которых таблицы предсказывающего анализатора строятся достаточно просто.

S-грамматикой будем называть такую контекстно-свободную грамматику, правые части правил которой начинаются с терминальных символов, причем для одного и того же левого символа правые части начинаются с разных символов.

Не S-грамматика:

$S \rightarrow aT$

$T \rightarrow bT$

$S \rightarrow TbS$

$T \rightarrow bT$

Аналогичная S-грамматика (распознает тоже):

$S \rightarrow abR$

$S \rightarrow bRbS$

$R \rightarrow a$

$R \rightarrow bR$

Таблица анализа для данной грамматики

Нетерминал	Входной символ	
	a	b
S	S:abR	S:bRbS
R	R:a	R:bR

Q-грамматика отличается от S-грамматики наличием аннулирующего правила (в правой части есть пустой символ) $\alpha \Rightarrow \varepsilon$.

1. $S \rightarrow aAS$
2. $S \rightarrow b$
3. $A \rightarrow cAS$
4. $A \rightarrow \varepsilon$

Для аннулирующих правил используется множество следующих символов. В данном случае FOLLOW(A)= {a,b}.

Таблица анализа для данной грамматики

Нетерминал	Входной символ		
	a	b	c
S	S:aAS	S:b	
A	A: ε	A: ε	A:cAS

LL(1) - грамматики. (left - leftmost)

LL(1) - грамматики относятся к нисходящим грамматикам (сверху - вниз).

Они отличаются от Q-грамматик тем, что правые части могут начинаться с нетерминальных символов, но таких, которые после

подстановок терминальных символов обеспечивают однозначность выбора грамматических правил.

В LL(k) - грамматиках разворачиваются самые левые нетерминальные символы сентенциальной формы, и анализируется очередной самый левый терминальный входной строки. Если для выбора правила подстановки достаточно анализ k самых левых символов входной строки, грамматику называют LL(k) - грамматикой. Но, поскольку грамматики LL(k) и LL(1) являются эквивалентными в плане порождаемых языков, остановимся на рассмотрении только последней.

Для LL(1) грамматик, таблицы анализа не имеют неоднозначно определенных входов.

1. $S \rightarrow AbB$ FIRST(AbB) = {a, b, c, e}
2. $S \rightarrow d$ FIRST(d)= {d}
3. $A \rightarrow CAb$ FIRST(CAb) = {a, e}
4. $A \rightarrow B$ FIRST (B) = {c, b}
5. $B \rightarrow cSd$ FIRST(cSd) = {c}
6. $B \rightarrow \varepsilon$ FOLLOW(B) ={b, d, $}
7. $C \rightarrow a$ FIRST (a) = {a}
8. $C \rightarrow ed$ FIRST (ed) = {e}

Таблично-управляемый предсказывающий анализатор

Магазинная память (стек) реализует память, работающий по принципу "последним прибыл, первым обслужен" (LIFO).

Основные операции, связанные со стеком:

- перация записи в стек заданной строки **push** ("..."). Строка помещается в стек, начиная с последнего символа, то есть первый символ оказывается в вершине стека;
- операция выталкивания символа из стека **pop**().

Анализатор на основе стека и таблицы анализа работает следующим образом. Она рассматривает X - символ на верхушке магазина и a - текущий входной символ. Эти два символа определяют действие анализатора. Имеются три возможности.

1. Если X=a=$, анализатор останавливается и сообщает об успешном окончании разбора.
2. Если X=a#$, анализатор удаляет X из магазина и продвигает указатель входа на следующий входной символ.
3. Если X - нетерминал, анализатор заглядывает в таблицу M[X,a]. По этому входу хранится либо правило для X, либо ошибка. Если, например, M[X,a]={ X-> UVW }, то есть правая часть правила X-> UVW, анализатор заменяет X на верхушке магазина на WVU {на верхушке U}. Будем считать, что анализатор в качестве выхода просто печатает использованные правила вывода. Если M[X,a]=error, анализатор прекращает работу и выдает сообщение об ошибке.

Упражнения и задания

Для заданных грамматик определить вид порождаемых цепочек, построить нисходящий распознаватель с определением множеств выбирающих символов и продемонстрировать принципы его работы на заданном или предложенном примере.

1) S:aSb S:bA A:aA A:
2) S:/*A*/ A:+A A:SA A:
3) S:aB+S S: B:aB B:
4) U:SU U: S:[U]
5) E:TM M:+E M:-E M: T:[T] T:
6) U:SU U: S:aA A:bA A:
7) Z:U U:LU U: L:Sbc S:aS S:
8) Z:U U:EU U: E:aSb S:aSb S:
9) Z:UM M:,UM M: U:SN N:SN N: S:a S:b S:c
10) Z:N N:UM M:,UM M: U:aSK S:aS S: K:[N] K:
11) U:ST T:,ST T: S:BA B:*B B: A:aA A:
12) U:aU U:{T}U U: T:bT T:{S}T T: S:cS S:
13) U:aA A:aA A: A:bB B:bB B:cC C:cC C:bB B:aA
14) T:UT T: U:<A A:U> A:-T-> A:>
15) S:APA P:+,- A:a,b

6. LR – грамматики (left - rightmost)

Сущность восходящего разбора

Восходящие методы синтаксического анализа состоят в том, что в цепочке (промежуточной или терминальной) ищется правая часть очередного правила, которое должно быть заменено своим нетерминалом. Очевидно, что не любая правая часть правила может быть свернута в любое время. Если посмотреть на дерево синтаксического разбора, то нетрудно заметить, что свертку необходимо производить регулярно, путем обхода дерева слева направо.

Основа - это полная правая часть первого правила при просмотре цепочки слева направо. Найденная в цепочке основа, должна быть заменена на нетерминал из левой части соответствующего правила.

Способ нахождения основы цепочки реализован в методе восходящего разбора "свертка-перенос":

- в процессе анализа используется магазинный автомат (КА со стеком);
-на каждом шаге работы автомата сравнивается пара символов - текущий символ входной цепочки и текущий символ в стеке. По результату сравнения возможны два действия:
- перенос терминального символа из строки в стек:

- свертка правой части правила, которое в данный момент находится в стеке. Возможны два принципиально различных варианта свертки: правило единственное, либо возможен выбор одного из нескольких.

Для формального задания поведения МА необходимо задать таблицу его действий на полный набор сочетаний - входной терминальный символ и любой символ в стеке. В клетке таблицы могут быть указаны следующие действия:

- сообщение о синтаксической ошибке;
- сообщение об успешном завершении разбора;
- перенос символа из входной цепочки в стек;
- вызов процедуры для анализа правой части правила в стеке и замены ее на нетерминал.

Для грамматики арифметических выражений со скобками приведем пример такой таблицы.

	ac	+-	*/	(	)	#
ac	-	C3	C3	-	C3	C3
+-	П	-	-	П	-	-
*/	П	-	-	П	-	-
(	П	-	-	П	-	-
)	-	C3	C3	C3	C3	C3
F	-	C2	C2	-	C2	C2
T	-	C1	П	-	C1	C1

- начало/конец цепочки

Процедуры свертки правил:

C1 = E→ E+T | E-T | T

C2 = T→ T*F | T/F | F

C3 = F→ (E) | a | c

Как видно, для простых грамматик построение такой таблицы не выходит за рамки здравого смысла. Для каждой пары символов производится анализ ситуации (контекста цепочки), в которой она может встретиться, на основании чего и заполняется клетка таблицы. Если ситуация свидетельствует, что не вся правая часть правила в стеке, то необходимо сделать перенос, если вся - анализировать возможные правила. На самом деле, таблицу можно упростить еще больше. При выполнении свертки группу правил с одинаковой правой частью (C1,C2,C3,C4) можно однозначно определить по сочетанию последнего (левого) символа правой части и текущего символа входной строки. Поэтому в таблице достаточно оставить только признак свертки.

LR грамматики

Эти грамматики относятся к восходящим грамматикам (снизу - вверх). В LR- грамматиках сворачиваются самые правые части правил для самых левых нетерминальных символов и анализируется очередной самый правый

символ свертываемой части строки. Грамматика называется LR(k) грамматикой, если при детерминированном разборе сверху вниз решение о свертке или сдвиге можно принять пуктем анализа k символов непросмотренной части входной части. При k=1 грамматика называется просто LR грамматикой. Для LR грамматики можно построить однозначную таблицу действий анализатора на основе магазинного автомата.

Грамматики предшествования

К числу LR- грамматик относятся *грамматики с предшествованием.* Определим специальные отношения, которые могут возникать между символами стоящими рядом в сентенциальной форме. Здесь правые части грамматических правил будем называть *свертками.*

1. Если S_i и S_j - два рядом стоящие символа входят в одну свертку, то между ними существует отношение : $S_i = *S_j$ (назовем его *равно*);

... S_i S_j...

Пример. В сентенциальной форме AbCdEfg при наличии правила K→CdE, существуют отношения

C =* d , d =* E

2. Если S_i и S_j два рядом стоящие символа и с S_j начинается какая-то свертка, то между ними существует отношение: $S_i <* \cdot S_j$;

S_i S_j...

Пример. В сентенциальной форме AbCdEfg при наличии правила L → dE Существует отношение

C <* d

3. а) Если S_i и S_j два рядом стоящие символа и S_i самый правый символ в свертке, то между ними существует отношение : $S_i *> S_j$;

... S_i S_j

Пример. В сентенциальной форме AbCdEfg при наличии правила L → dE Существует отношение

E *> f

б) Если S_i и S_j два рядом стоящие символа и S_i самый правый символ в одной свертке, а S_j - самый левый в другой, то между ними существует отношение : $S_i *> S_j$;

... S_i S_j...

Пример. В сентенциальной форме AbCdEfg при наличии правил L → dE и M → fg

Существует отношение E *> f

Для удобства дальнейшей работы составим таблицу *левых и правых* символов, которые могут оказаться в подставленных вместо этих символов цепочках на месте данных нетерминальных символов. Таблица строится на основе анализа грамматических правил.

A → BC

B → lC
B → CA
C → d

Выявим отношения:

B =* C l =* C C =* A
B<* · Л(C) (множество левых для C)
B<* · Л(C) = {d}
l<* · Л(C)
C<* · Л(C) = {B, l, C, d}

{C, A, d} = П(B) · *>C
0 = П(l) · *>C
{d} = П(C) · *>C (3a)
{C, A, d} = П(B) · *> Л(C) = {d} (3b)
{d} = П(C) · *>Л(A)={B, l, C, d}

И сведем их в таблицу - *матрицу предшествования.*

	A	B	C	d	l
A			· *>	· *>	
B			=*	<* ·	
C	=* ·	<*	*><*	*> <*	<*
d	*>	*>	*>	· *>	*>
l			=* ·	<* ·	

Грамматика называется **грамматикой с предшествованиями**, если между любыми двумя символами, стоящими рядом в сентенциальной форме, существует строго одно отношение предшествования.

Использование матриц с предшествованием.

S → BC
B → Axz
C → xx
A → xy

Считаем, что она построена.

Использование матрицы:

xyxzxx - проставляем все значки

├<* x =* y · *> x =* z · *> x =* x *>┤
├<* A · x · y · *> x · x · *>┤
├<* B <* x =* x *>┤
├<* B =* C *>┤
├S┤

Алгоритм распознавания:

1. Между символами строки вставляются отношения предшествования.
2. Строка просматривается слева направо до первого символа $\cdot *>$, после этого просмотр идет в обратном направлении до первого встречаемого символа $*<\cdot$ - между этими символами и находится свертка, то есть правая часть правила, которая заменяется левой.
3. Восстанавливаются отношения предшествования.
4. Возвращение к первому пункту. Процесс продолжается до получения начального нетерминального символа. Если этот процесс не завершится успешно, строка не принадлежит данной грамматике.

У этого метода есть минусы:

1. Далеко не во всех случаях удается построить грамматику с предшествованиями.
2. На практике символов может быть много сотен сотни и в результате получается слабозаполненная матрица большой размерности.

Функции предшествования

Этот интересный метод придумал Р.Флойд – автор многих остроумных решений в программировании. Вместо матрицы строятся две специальные функции f и g , такие что:

1. Если $S_i \cdot *> S_j \Rightarrow f(S_i) > g(S_j)$.
2. Если $S_i <* S_j \Rightarrow f(S_i) < g(S_j)$.
3. Если $S_i =* S_j \Rightarrow f(S_i) = g(S_j)$.

Тогда, вместо поиска с помощью матрицы отношения предшествования между символами, просто происходит сравнение числовых значений соответствующих функций на больше, меньше, равно.

Построение функций предшествования:

0. Строится матрица предшествования и начальные значения функций принимаются равными единице: $f(S_i) = g(S_j) = 1$.
1. Матрица просматривается по строкам в поисках отношений $\cdot *>$ и, если $f(S_i) > g(S_j)$, то идем дальше, если же $S_i *> S_j$, а $f(S_i) \le g(S_j)$, то увеличиваем значение $f(S_i) = f(S_i) = g(S_j) + 1$.
2. Матрица просматривается по столбцам в поисках отношений $<* \cdot$ и, если $f(S_i) < g(S_j)$, то идем дальше, если же $S_i <* S_j$, а $f(S_i) \ge g(S_j)$, то увеличиваем значение $g(S_j) = g(S_j) = f(S_i) + 1$.
3. Матрица просматривается в поисках отношений $=*$ и, если $f(S_i) = g(S_j)$, то идем дальше, если $S_i =* S_j$, а $f(S_i) \ne g(S_j)$, то выравниваем значения функций путем увеличения меньшего из значений до большего - $f(S_i) = g(S_j) = \max[f(S_i), g(S_j)]$.
4. Возвращение к первому пункту.

Повторять до тех пор, пока рост функций не прекратится (или когда значение одной из функций не превысит 2*n, где n - размерность матрицы - в этом случае алгоритм не сходится).

Однако этот метод не свободен от недостатков:
1. Алгоритм не всегда сходится (не всегда приводит к построению функций).
2. При переходе к функциям происходит «незаконное доопределение» матрицы. То есть как бы появляются отношения предшествования между парами символов, для которых в исходной матрице отношение отсутствовало.

Грамматики операторного предшествования

Если в правилах приведенной обратимой грамматики не встречаются рядом два нетерминала, говорят, что грамматика является операторной. Классический пример - грамматика арифметических формул. Для таких грамматик можно вычислять отношения предшествования только на множестве терминальных символов. Для этого модифицируем правила вычисления отношений предшествования:

a =' b, если в грамматике имеется правило A : ...ab...
или A : ...aBb...

a <' b, если в грамматике имеется правило A : ...aB...
и B :: b...
или B :: Cb...

a >' b, если в грамматике имеется правило A : ...Bb...
и B :: ...a
или B :: ...aC

<' a, если в грамматике имеется вывод S :: Ca...
или S :: a...

a >' #, если в грамматике имеется вывод S :: ...aC
или S :: ...a

Если между любыми двумя терминалами выполняется не более одного отношения предшествования, операторная грамматика называется грамматикой операторного предшествования.

Эти отношения позволяют определять терминалы, входящие в правую часть сворачиваемого правила, но не нетерминалы. Однако при практическом разборе формул нет необходимости различать нетерминалы S, T и E. Они были введены только для придания грамматике однозначности и учета приоритета и ассоциативности операций. Теперь, получив отношения предшествования, можно вновь заменить эти искуственно введенные нетерминалы на S:

S : S+S | S*S | (S) | a

и получить разбор, обрабатывая все нетерминалы одинаково.

Упражнения

Для заданных грамматик определить вид порождаемых цепочек, построить распознаватель для метода "свертка-перенос" и продемонстрировать принципы его работы на заданном или предложенном примере.

1) U:S U:US S:aA A:bA A:b
2) E:E+T E:E-T E:T T:[] T:[T]
3) U:US U:S S:A A:aA A:ab
4) U:US U:S S:[] S:[U]
5) Z:E E:E,T E:T T:a T:b T:Ta T:Tb
6) Z:U U:UE U:E E:Abc E:bc A:Aa A:a
7) Z:U U:UE U:E E:ab E:aEb
8) Z:E E:E,T E:T T:S[E] T:S S:Sa S:a
9) U:U,T U:T T:BA T:A B:B* B:* A:Aa A:a
10) U:U,T U:T T:*T T:A A:Aa A:a
11) U:UT U:T T:a T:{S} S:SR S:R R:b R:{M} M:Mc M:c
12) A:Aa B:Ab B:Bb C:Bc C:Cc B:Cb A:Ba A:a
13) U:ABC A:Aa A:a B:Bb B:b C:Cc C:c
14) T:TU T:U U:<U> U:<-T-> U:<>
15) S:AS,SB,AB A:a B:b

7. Генерация выходного текста

Польская инверсная запись

При рассмотрении вопросов генерации выходного текста надо иметь в виду то, что реально выходной текст программы после трансляции - это, как правило, не выполняемый код, а некоторая промежуточная форма, поскольку программа в дальнейшем может быть загружена для выполнения в разные места памяти и т.д. С другой стороны, выполняемая программа (или программа в близком к такой форме виде) машинно-зависима, то есть использует конкретную систему команд и другие конкретные архитектурные особенности. Так что процесс генерации - трудоемкий процесс, требующий большой и весьма кропотливой работ. Поэтому рассмотрим лишь основную идею генерации и остановимся на использовании в вычислениях постфиксных представлений.

Генерация

Существует три способа записи операций:

1. Инфиксный: a*b (например, a + b).
2. Префиксный: *ab (например, Σ(a,b)).
3. Постфиксный: ab* (например, a и b - просуммировать)

Часто для арифметических выражений выполняют преобразование в **польскую инверсную запись (ПОЛИЗ)**, что упрощает последующее выполнение-вычисление.

Данная запись названа польской в память о польском математике Яне Лукашевиче.

Алгоритм преобразования арифметических выражений в ПОЛИЗ

1. Поступающие на вход операнды сразу проходят на выход.

2. Поступающие на вход операторы сравниваются по приоритету. Если приоритет оператора на входе магазина больше, чем в верхушке магазина, то оператор со входа поступает в магазин (first in/last out). Если приоритет оператора на входе меньше или равен приоритету оператора в верхушке магазина, то оператор из верхушки магазина идет на выход и сравнение повторяется. Эти процедуры выполняются до тех пор, пока не исчерпается строка.

Операции	Магазинный	Сравнительный
(	∅	∞
:=	∅	∞
+ -	1	1
* /	2	2
(степень) ↑	3	3
)	-	∅

Трехадресный код

Трехадресный код- это последовательность операторов вида x:= y op z, где x,y и z - имена, константы или сгенерированные компилятором временные объекты. Здесь op - двуместная операция, например, операция плавающей или фиксированной арифметики, логическая или побитовая. В правую часть может входить только один знак операции.
Составные выражения должны быть разбиты на подвыражения, при этом могут появиться временные имена (переменные). Смысл термина "трехадресный код" в том, что каждый оператор обычно имеет три адреса: два для операндов и один для результата. Например, выражение x+y*z может быть протранслировано в последовательность операторов

t1:=y*z
t2:=x+t1

где t1 и t2 - имена, сгенерированные компилятором. В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления программы и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться. Например, условный оператор

if A>B then S1 else S2

может быть представлен следующим кодом:

t:=A-B
JGT t,S2
.....

Здесь JGT - двуместная операция условного перехода, не вырабатывающая результата. Разбиение арифметических выражений и операторов управления делает трехадресный код удобным при генерации машинного кода и оптимизации. Использование имен промежуточных

значений, вычисляемых в программе, позволяет легко переупорядочивать трехадресный код.

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t1
a := t5
```

Трехадресный код - это абстрактная форма промежуточного кода. В реализации трехадресный код может быть представлен записями с полями для операций и операндов. Рассмотрим три реализации трехадресного кода: четверки, тройки и косвенные тройки.

Четверка - это запись с четырьмя полями, которые будем называть op, arg1, arg2 и result. Поле op содержит код операции. В операторах с унарными операциями типа x:=-y или x:=y arg2 не используется. В некоторых операциях (типа "передать параметр") могут не использоваться ни arg2, ни result. Условные и безусловные переходы помещают в result метку перехода. Обычно содержимое полей arg1, arg2 и result - это указатели на входы таблицы символов для имен, представляемых этими полями. Временные имена вносятся в таблицу символов по мере их генерации.

Чтобы избежать внесения новых имен в таблицу символов, на временное значение можно ссылаться, используя позицию вычисляющего его оператора. В этом случае трехадресные операторы могут быть представлены записями только с тремя полями: op, arg1 и arg2. Поля arg1 и arg2 - это либо указатели на таблицу символов (для имен, определенных программистом, или констант), либо указатели на тройки (для временных значений). Такой способ представления трехадресного кода называют тройками. Тройки соответствуют представлению синтаксического дерева или ОАГ с помощью массива вершин.

Числа в скобках - это указатели на тройки, а имена – это указатели на таблицу символов. На практике информация, необходимая для интерпретации различного типа входов в поля arg1 и arg2, кодируется в поле op или дополнительных полях.

Трехадресный код может быть представлен не списком троек, а списком указателей на них. Такая реализация обычно называется косвенными тройками.

При генерации объектного кода каждой переменной, как временной, так и определенной в исходной программе, назначается память периода исполнения, адрес которой обычно хранится в таблице генератора кода. При использовании четверок этот адрес легко получить через эту таблицу. Более существенно преимущество четверок проявляется в оптимизирующих компиляторах, когда может возникнуть необходимость перемещать операторы. Если перемещается оператор, вычисляющий x, не требуется изменений в операторе,

использующем x. В записи же тройками перемещение оператора, определяющего временное значение, требует изменения всех ссылок на этот оператор в массивах arg1 и arg2. Из-за этого тройки трудно использовать в оптимизирующих компиляторах. В случае применения косвенных троек оператор может быть перемещен переупорядочиванием списка операторов. При этом не надо менять указатели на op, arg1 и arg2. Этим косвенные тройки похожи на четверки. Кроме того, эти два способа требуют примерно одинаковой памяти. Как и в случае простых троек, при использовании косвенных троек выделение памяти для временных значений может быть отложено на этап генерации кода. По сравнению с четверками при использовании косвенных троек можно сэкономить память, если одно и то же временное значение используется, более одного раза.

Упражнения

1. Представить в ПОЛИЗе следующие выражения:

a) a+b-c b) a∗b+c/a

c) a/(b+c)∗a d) (a+b)/(c+a∗b)

e) a and b or c f) not a or b and a

g) x+y=x/y h) (x∗x+y∗y < 1) and (x > 0)

2. Для следующих выражений в ПОЛИЗе дать обычную инфиксную запись:

a) ab∗c b) abc∗/ c) ab+c∗

d) ab+bc-/a+ e) a not b and not f) abca and or and

g) 2x+2x∗<

3. Используя стек, вычислить следующие выражения в ПОЛИЗе:

a) xy∗xy/+ при x = 8, y = 2 ;

b) a2+b/b4∗+ при a = 4, b = 3 ;

c) ab not and a or not при a = b = true ;

d) xy∗0 > y2x- < and при x = y = 1 .

4. Записать в ПОЛИЗе следующие операторы языка Си и, используя стек, выполнить их при указанных начальных значениях переменных:

a) if (x != y) x = x+1 ; при x = 3 ;

b) if (x > y) x = y ; else y = x ; при x = 5, y = 7;

c) while (b > a) {b = b-a;} ; при a = 3, b = 7;

d) do {x = y; y = 2;} while (y > 9); при y = 2;

e) S = 0; for (i = 1; i <= k; i = i + 1) {S = S + i∗i;} при k = 3 ;

f) switch (k) {

case 1: a = not a; break;

case 2: b = a or not b ;

case 3: a = b ;

}
при k = 2, a = b = fal... .

5. Используя стек, выполнить следующие действия, записанные в ПОЛИЗе, при x = 9, y = 15 (считаем,что элементы ПОЛИЗа перенумерованы с 1).

z, x, y, *, :=, x, y, <>, 30, !F, x, y, <, 23, !F, y, y, x, -, :=, 6, !, x, x, y, -, :=, 6, !, z, z, x, /, :=

Описать заданные действия на Си, не используя оператор goto.

6. Записать в ПОЛИЗе следующие операторы Паскаля:

a) for I := E1 to E2 do S
b) case E of
c1: S1;
c2: S2;
....
cn: Sn
end;
c) repeat S1; S2; ... ;Sn until B;

7. Записать в ПОЛИЗе следующие фрагменты программ на Паскале:

a) case k of
1: begin a:=not(a or b and c); b:=a and c or b end;
2: begin a:=a and (b or not c); b:= not a end;
3: begin a:=b or c or not a; b:==b and c or a end
end
b) S:=0; for i:=1 to N do
begin d:=i*2; a:=a+d*((i-1)*N+5)
S:=-a*d+S
end
c) c:=a*b; while a<>b do
if a<b then b:=b-a else a:=a-b;
c:=c/a

8. Предоставить в виде четверки следующие выражения:

a) a+b-c b) a*b+c/a
c) a/(b+c)*a d) (a+b)/(c+a*b)
e) a and b or c f) not a or b and a
g) x+y=x/y h) (x*x+y*y < 1) and (x > 0)

9. Представить в виде тройки следующие выражения:

a) ab*c b) abc*/ c) ab+c*
d) ab+bc-/a+ e) a not b and not f) abca and or and
g) 2x+2x*<

10. Представить в виде косвенной тройки следующие выражения и вычислить:

a) xy∗xy/+ при x = 8, y = 2 ;

b) a2+b/b4∗+ при a = 4, b = 3 ;

c) ab not and a or not при a = b = true ;

d) xy∗0 > y2x- < and при x = y = 1 .

11. Записать в трехадресном коде следующие операторы языка Си и выполнить их при указанных начальных значениях переменных:

a) if (x != y) x = x+1 ; при x = 3 ;

b) if (x > y) x = y ; else y = x ; при x = 5, y = 7;

c) while (b > a) {b = b-a;} ; при a = 3, b = 7;

d) do {x = y; y = 2;} while (y > 9); при y = 2;

e) S = 0; for (i = 1; i <= k; i = i + 1) {S = S + i∗i;} при k = 3 ;

f) switch (k) {
case 1: a = not a; break;
case 2: b = a or not b ;
case 3: a = b ;
}
при k = 2, a = b = false .

12. Записать в трехадресном виде следующие операторы Паскаля:

a) for I := E1 to E2 do S

b) case E of
c1: S1;
c2: S2;
....
cn: Sn
end;

c) repeat S1; S2; ... ;Sn until B;

13. Записать в виде четверки следующий фрагмент программы на Паскале:

case k of
1: begin a:=not(a or b and c); b:=a and c or b end;
2: begin a:=a and (b or not c); b:= not a end;
3: begin a:=b or c or not a; b:=b and c or a end
end

14. Записать в виде тройки следующий фрагмент программы на Паскале:

S:=0; for i:=1 to N do

begin d:=i∗2; a:=a+d∗((i-1)∗N+5)

S:=-a∗d+S
end

15. Записать в виде косвенной тройки следующий фрагмент программы на Паскале:

```
c:=a*b; while a<>b do
if a<b then b:=b-a else a:=a-b;
c:=c/a
```

8. Элементы теории перевода

Синтаксически управляемый перевод

Схемой синтаксически управляемого перевода (или трансляции, сокращенно: СУ-схемой) называется пятерка Tr=(N,T,П,R,S), где

N - конечное множество нетерминальных символов;

T - конечный входной алфавит;

П - конечный выходной алфавит;

R - конечное множество правил перевода вида A->u, A1=v1, A2=v2, ... , Am=vm, удовлетворяющих следующим условиям:

- каждый символ, входящий в v1, ..., vm, либо принадлежит П, либо является Bk для B<-N и B входит в v (здесь k - номер вхождения B в v),

- если u имеет более одного вхождения символа B, то каждый символ Bk во всех v соотнесен (верхним индексом) с конкретным вхождением B;

S - начальный символ, выделенный нетерминал из N.

A->u называют входным правилом вывода, Ai - переводом нетерминала A и Ai=vi - элементом перевода, связанным с этим правилом перевода. Если через P обозначить множество входных правил вывода всех правил перевода, то G=(N,T,P,S) будет входной грамматикой для Tr. Если в СУ-схеме Tr нет двух правил перевода с одинаковым входным правилом вывода, то ее называют семантически однозначной. Выход СУ-схемы определим снизу вверх. С каждой внутренней вершиной n дерева разбора (во входной грамматике), помеченной A, свяжем одну цепочку для каждого Ai. Эта цепочка называется значением (или переводом) символа Ai в вершине n. Каждое значение вычисляется подстановкой значений символов перевода данного элемента перевода Ai=vi, определенных в прямых потомках вершины n. Переводом т(Tr), определяемым СУ-схемой Tr, назовем множество {(x,y)|x имеет дерево разбора во входной грамматике для Tr и y - значение выделенного символа перевода S в корне этого дерева}. Если Tr=(N,T,П,R,S) - СУ-схема, то т(Tr) называется синтаксически управляемым переводом (СУ-переводом).

Рассмотрим формальное дифференцирование выражений, включающих константы 0 и 1, переменную x и функции sin, cos, + и *. Такие выражения порождает грамматика

```
E -> E+T | T
T -> T*F | F
```

F -> (E) | sin(E) | cos(E) | x | 0 | 1

Свяжем с каждым из E, T и F два перевода, обозначенных индексом 1 и 2. Индекс 1 указывает на то, что выражение не дифференцировано. 2 - что выражение продифференцировано. Формальная производная - это E2. Законы дифференцирования таковы:

d(f(x)+g(x))=df(x)+dg(x)
d(f(x)*g(x))=f(x)*dg(x)+g(x)*df(x)
dsin(f(x))=cos(f(x))*df(x)
dcos(f(x))=-sin(f(x))df(x)
dx=1
d0=0
d1=0

Эти законы реализуются СУ-схемой:

E -> E+T, E1=E1+T1, E2=E2+T2
E -> T, E1=T1, E2=T2
T -> T*F, T1=T1*F1, T2=T1*F2+T2*F1
F -> (E), F1=(E1), F2=(E2)
F -> sin(E), F1=sin(E1), F2=cos(E1)*(E2)
F -> cos(E), F1=cos(E1),F2=-sin(E1)*(E2)
F -> x, F1=x, F2=1
F -> 0, F1=0 F2=0
F -> 1, F1=1 F2=0

Теорема 1. Если число вхождений каждого нетерминала в слове v не превосходит 1, то t(Tr) является КС-языком. Обратное не всегда верно.

Пример. T=({S,A},{a},{a,b},{S->A,AbAbA;A->a,a;A->aA,aA}.

Здесь входной язык {an|n>=1}, выходной {anbanban}. Выходной язык не КС.

Теорема 2. Для каждого магазинного преобразователя существует эквивалентная СУ-схема.

Обратное, вообще говоря, неверно.

Определение. Семантически однозначная СУ-схема Tr=(N,T,П,R,S) называется простой, если для каждого правила A->u,v из R соответствующие друг другу вхождения нетерминалов встречаются в u и v в одном и том же порядке.

Перевод, определяемый простой СУ-схемой, называется простым **синтаксически управляемым переводом (простым СУ-переводом).**

Теорема 3. Пусть Tr=(N,T,П,R,S) - простая СУ-схема. Существует такой МП-преобразователь P, что t(P)=t(Tr).

Таким образом, класс трансляций, определяемых магазинными преобразователями, совпадает с классом простых СУ-переводов.

Теорема 4. Пусть Tr=(N,T,П,R,S) - семантически однозначная простая СУ-схема, входной грамматикой которой служит LL(k)- грамматика. Тогда перевод {x$,y)|(x,y)<-t(Tr)} можно осуществить детерминированным МП-преобразователем.

Существуют семантически однозначные простые СУ-схемы, имеющие в качестве входных грамматик LR(k) грамматики и нереализуемые ни на каком ДМП-преобразователе.

Пример. Рассмотрим простую СУ-схему Т с правилами

S -> Sa, aSa

S -> Sb, bSb

S -> e, e

Входная грамматика является LR(1) грамматикой, но не существует ДМП-преобразователя, определяющего перевод

{(x$,y)|(x,y)<-t(Tr)} [5].

Определение. Назовем СУ-схему Tr=(N,T,П,R,S) постфиксной, если каждое правило из R имеет вид A->u,v, где v<-N*П*.

Иными словами, каждый элемент перевода представляет собой цепочку из нетерминалов, за которыми следует цепочка выходных символов.

Теорема 5. Пусть Tr=(N,T,П,R,S) - семантически однозначная простая постфиксная СУ-схема, входной грамматикой которой служит LR(k)-грамматика. Тогда перевод {(x$,y)|(x,y)<-t(Tr)} можно осуществить детерминированным МП-преобразователем.

Атрибутные грамматики

Среди всех формальных методов описания языков программирования атрибутные грамматики получили, по-видимому, наибольшую известность и распространение. Причиной этого является то, что формализм атрибутных грамматик основывается на дереве разбора программы в КС-грамматике, что сближает его с хорошо разработанной теорией и практикой построения трансляторов.

Определение атрибутных грамматик

Пусть G - КС-грамматика: G=(T,N,P,Z), где T, N, P, Z, - соответственно, множество терминальных символов, нетерминальных символов, множество правил вывода и аксиома грамматики. Правила вывода КС-грамматики будем записывать в виде

p: X0 -> X1 ... Xn

и будем предполагать, что G - редуцированная КС-грамматика, т.е. в ней нет нетерминальных символов, для которых не существует полного дерева вывода, в которое входит этот нетерминал.

С каждым символом X∈ N U T свяжем множество A(X) атрибутов символа X. Некоторые из множеств A(x) могут быть пусты. Запись a(X) означает, что a ∈A(X).

С каждым правилом вывода p <- P свяжем множество F семантических правил, имеющих следующую форму:

a0 = fpa0(a1, ... , aj),

где ik <- [0,np] - номер символа правила p, а ak – атрибут символа Xik , т.е. ak <- A(Xik).

В таком случае будем говорить, что a0 "зависит" от a1,...,aj или что a0 "вычисляется по" a1,...,aj. В частном случае j может быть равно нулю, тогда будем говорить, что атрибут a0 "получает в качестве значения константу".

КС-грамматику, каждому символу которой сопоставлено множество атрибутов, а каждому правилу вывода - множество семантических правил, будем называть атрибутной грамматикой (AG).
Назовем атрибут a(X0) синтезируемым, если одному из правил вывода p: X0 ->X1 ... Xnp сопоставлено семантическое правило a<0>=fa<0>(...).
Назовем атрибут a(Xi) наследуемым, если одному из правил вывода p:X0 -> X1 ... Xi ... Xnp сопоставлено семантическое правило a=fa(...), i <- [1,np].
Множество синтезируемых атрибутов символа X обозначим через S(X), наследуемых атрибутов - через I(X).

Будем считать, что значение атрибутов терминальных символов - константы, т.е. их значения определены, но для них нет семантических правил, определеяющих их значения.

Атрибутированное дерево разбора

Если дана атрибутная грамматика AG и цепочка, принадлежашая языку, определяемому G, то можно построить дерево разбора этой цепочки в грамматике G. В этом дереве каждая вершина помечена символом грамматики G. Припишем теперь каждой вершине множество атрибутов, сопоставленных символу, которым помечена эта вершина. Атрибуты, сопоставленные вхождениям символов в дерево разбора, будем называть вхождениями атрибутов в дерево разбора, а дерево с сопоставленными каждой вершине атрибутами - атрибутированным деревом разбора.

Между вхождениями атрибутов в дерево разбора существуют зависимости, определяемые семантическими правилами, соответствующими примененным синтаксическим правилам.

Язык описания атрибутных грамматик

Формализм атрибутных грамматик оказался очень удобным средством для описания семантики языков программирования.
Вместе с тем выяснилось, что реализация вычислителей для атрибутных грамматик общего вида сталкивается с большими трудностями. В связи с этим было сделано множество попыток рассматривать те или иные классы атрибутных грамматик, обладающих "хорошими" свойствами. К числу таких свойств относятся прежде всего простота алгоритма проверки атрибутной грамматики на зацикленность и простота алгоритма вычисления атрибутов для атрибутных грамматик данного класса.

Атрибутные грамматики использовались для описания семантики языков программирования и было создано несколько систем автоматизации разработки трансляторов, основанных на формализме атрибутных грамматик. Опыт их использования показал, что "чистый" атрибутный

формализм может быть успешно применен для описания семантики языка, но его использование вызывает трудности при создании транслятора. Эти трудности связаны как с самим формализмом, так и с некоторыми технологическими проблемами. К трудностям первого рода можно отнести несоответствие чисто функциональной природы атрибутного вычислителя и связанной с ней неупорядоченностью процесса вычисления атрибутов (что в значительной степени является преимуществом этого формализма) и упорядоченностью элементов программы. Это несоответствие ведет к тому, что приходится идти на искусственные приемы для их сочетания. Технологические трудности связаны с эффективностью трансляторов, генерированных с помощью атрибутных систем. Как правило, качество таких трансляторов довольно низко из-за больших расходов памяти, неэффективности искусственных приемов, о которых было сказано выше.

Учитывая это, мы будем вести дальнейшее изложение на языке, сочетающем особенности атрибутного формализма и обычного языка, в котором предполагается возможность управления порядком исполнения операторов. Этот порядок привязывается к обходу атрибутированного дерева разбора сверху вниз, слева направо. Все атрибуты должны быть вычислены за один такой обход. Атрибутные грамматики, обладающие таким свойством, называются L-атрибутными.

При записи синтаксиса мы будем использовать расширенную БНФ.

Элемент правой части синтаксического правила, заключенный в скобки [ ], может отсутствовать. Элемент правой части синтаксического правила, заключенный в скобки ( ), означает возможность повторения один или более раз. Элемент правой части синтаксического правила, заключенный в скобки [()], означает возможность повторения ноль или более раз. В скобках [] или [()] может указываться разделитель конструкций.

Описание атрибутной грамматики состоит из раздела описания атрибутов и раздела правил. Раздел описания атрибутов определяет состав атрибутов для каждого символа грамматики и тип каждого атрибута. Правила состоят из синтаксической и семантической части. В синтаксической части используется расширенная БНФ. Семантическая часть правила состоит из локальных объявлений и семантических действий. В качестве семантических действий допускаются как атрибутные присваивания, так и составные операторы.

Метка в семантической части правила привязывает выполнение оператора к обходу дерева разбора сверху-вниз слева направо. Конструкция "i : оператор" означает, что оператор должен быть выполнен сразу после обхода i-й компоненты правой части.

Конструкция "i E : оператор" означает, что оператор должен быть выполнен, только если порождение i-й компоненты правой части пусто. Конструкция "i A : оператор" означает, что оператор должен быть выполнен после разбора каждого повторения i-й компоненты правой части (имеется в виду конструкция повторения).

Каждое правило может иметь локальные определения (типов и переменных). В формулах используются как атрибуты символов данного правила (локальные атрибуты) и в этом случае соответствующие символы указываются номерами в правиле (0 для символа левой части, 1 для символа правой части и т.д.), так и атрибуты символов предков левой части правила (глобальные атрибуты). В этом случае соответствующий символ указывается именем нетерминала. Таким образом, на дереве образуются области видимости атрибутов: атрибут символа имеет область видимости, состоящую из правила, в которое символ входит в правую часть, плюс все поддерево, корнем которого является символ, за исключением поддеревьев - потомков того же символа в этом поддереве.

Упражнения

1. Написать грамматику для выражений, содержащих переменные, знаки операций +, -, *, / и скобки ( ), где операции должны выполняться строго слева направо, но приоритет скобок сохраняется. Определить действия по переводу таких выражений в ПОЛИЗ.
2. Изменить приоритет операций отношения в М - языке (сделать его наивысшим). Построить соответствующую грамматику, отражающую этот приоритет. Написать синтаксический анализатор, обеспечить контроль типов, задать перевод в ПОЛИЗ.
3. Написать КС-грамматику, аналогичную данной,

$E \rightarrow T\ \{+T\}$

$T \rightarrow F\ \{*F\}$

$F \rightarrow (E) \mid i$

с той лишь разницей, что в новом языке будет допускаться унарный минус перед идентификатором, имеющий наивысший приоритет (например, a*-b+-c допускается и означает a*(-b)+(-c). В созданную грамматику вставить действия по переводу такого выражения в ПОЛИЗ. Для каждой используемой процедуры привести ее текст на Си.
4. Дана грамматика, описывающая выражения:

$E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$ $\quad T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow PF'$ $\quad F' \rightarrow \hat{}PF' \mid \varepsilon$

$P \rightarrow (E) \mid i$

Включить в эту грамматику действия по переводу этих выражений в ПОЛИЗ. Для каждой используемой процедуры привести ее текст на Си.
5. Построить регулярную грамматику для языка L1, вставить в нее действия по переводу L1 в L2.

$L1 = \{1^m 0^n \mid n,m>0\}$

$L2 = \{1^{m-n} \mid$ если m>n;

$0 \mid$ если m<n;

ε | если m=n}

(Эта задача аналогична задаче выдачи сообщений об ошибке в балансе скобок).

6. Построить регулярную грамматику для языка L1, вставить в нее действия по переводу цепочек языка L1 в соответствующие цепочки языка L2.

$L1 = \{1^n 0^m 1^m 0^n \mid m,n > 0\}$.

$L2 = \{1^m 0^{n+m} \mid m,n > 0\}$

7. Построить регулярную грамматику для языка L1, вставить в нее действия по переводу цепочек языка L1 в соответствующие цепочки языка L2.

$L1 = \{b_i \mid b_i = (i)_2$, т.е. b_i -это двоичное представление числа $i \in N\}$

$L2 = \{(b_{i+1})^R \mid b_{i+1} = (i+1)_2, \omega^R$ - перевернутая цепочка $\omega\}$

8. Построить грамматику, описывающую целые двоичные числа (допускаются незначащие нули). Вставить в нее действия по переводу этих целых чисел в четверичную систему счисления.

9. Написать грамматику для выражений, содержащих переменные, знаки операций +, -, *, /, ** и скобки ( ) с обычным приоритетом операций и скобок. Включить в эту грамматику действия по переводу этих выражений в префиксную запись (операции предшествуют операндам). Предложить интерпретатор префиксной записи выражений.

10. В грамматику, описывающую выражения, включить действия по переводу выражений из инфиксной формы (операция между операндами) в функциональную запись.

Например,

a+b ==> + (a, b)

a+b*c ==> + (a, * (b, c))

Литература

Основная литература

1. Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов. –СПб: Питер, 2003.-396 с.
2. Афанасьев А.Н. Формальные языки и грамматики.: Учебная школа: УлГТУ, 1997. – 84 с.
3. Вирт Н. Алгоритмы и структуры данных – М.; МИР, 1989. – 360 с.
4. Гордеев А.З., Молчанов А.Д. Системное программное обеспечение. – Спб-Петер, 2002. -734с.
5. Дворогин А.И. Основы трансляций. Учебное пособие. – Волгоград. ВолГТУ,1999.80г.
6. Карпов С.Ю. Теория автоматов. Учебные пособия для вузов. –СПб: Питер, 2003.-201с.
7. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции -.М: Мир, 1979.-487с.

Дополнительная литература

1. Кнут Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы. - М.: Мир, 1976.
2. Грисс Д. Конструирование компиляторов для цифровых вычислительных машин. –М. Мир, 1975-544с.
3. Курочкин В.М. Алгоритм распределения регистров для выражений за один обход дерева вывода//2 Всес. конф "Автоматизация производства ППП и трансляторов". 1983. С.104-105.
4. Надежин Д.Ю., В.А.Серебряков, В.М.Ходукин. Промежуточный язык Лидер (предварительное сообщение) // Обработка символьной информации. - М., 1987. С. 50-63.
5. Aho A., Sethi R., Ullman J. Compilers: principles, techniques and tools. N.Y.: Addison-Wesley, 1986.
6. Бек Л.Введение в системное программирование. М,: Мир, 1988.-448 с.
7. Компаниец Р.И. и др. Системное программирование. Основы построения трансляторов.- СПб.: КОРОНА принт, 2000 г. -256 с.
8. Aho A.U.,Ganapathi M.,Tjiang S.W. Code generation using tree matching and dynamic programming // ACM Trans.Program. Languages and Systems.1989. V.11.N 4.
9. Fraser C.W., Hanson D.R. A Retargetable compiler for ANSI C. // SIGPLAN Notices. 1991. V 26.
10. A. Bezdushny, V. Serebriakov. The use of the parsing method for optimal code generation and common subexpression elimination // Techn. et Sci. Inform. 1993. V.12. N.1. P.69-92.
11.Harrison M.A. Introduction to formal language theory. Reading, Mass.: Addison-Wesley, 1978.

ОГЛАВЛЕНИЕ

Языки и грамматики.....3
Формальные грамматики.....7
Регулярные языки и грамматики.....11
КС – Грамматики.....16
LL грамматики.....24
LR – грамматики.....29
Генерация выходного текста.....35
Элементы теории перевода.....41
Список литературы.....48
Оглавление.....49

Методическое пособие для практических занятий по дисциплине «Системное программное обеспечение»

Разработка рассмотрена на заседании кафедры «ТП»
и рекомендована к печати
(протокол № от 2008 года)

Авторы — Зав. каф. Назиров Ш.А.
Доц.каф. Кабулов Р.В.
Ст.пр. каф. Уринбаев С.К.

Ответственный редактор — Проректор по учебной работе ТУИТ, д.т.н., проф. Каримов М.М.

Корректор — Павлова С.И.

Бумага офсетная. Заказ № 189
Тираж. 50
Отпечатано в типографии ТУИТ
Ташкент 700084, ул.А.Тимура - 108