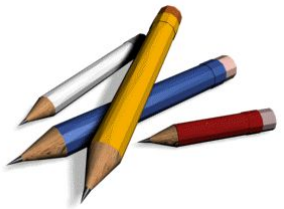
Синтаксический анализ

Синтаксический анализ

Синтаксический анализ (распознавание, разбор, parsing) является обязательной фазой в работе любого транслятора.

Синтаксический анализ преследует две цели:

– определить принадлежность цепочки символов языку;
– выявить структуру этой цепочки.

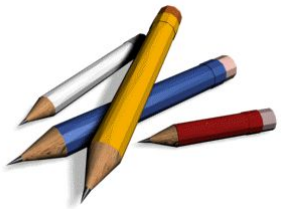


Синтаксический анализ

Структуру цепочки обычно представляют с помощью графа, называемого *синтаксическим деревом* (деревом вывода, деревом разбора).

Термин *дерево вывода* указывает на то, что данный граф отображает последовательность вывода цепочки символов из начального символа грамматики.

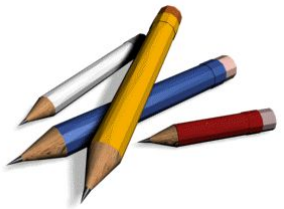
Термин *дерево разбора* указывает на то, что данное дерево строится в процессе анализа (разбора) заранее заданной цепочки.



Синтаксический анализ

Успешное восстановление дерева вывода для заданной цепочки означает, что цепочка есть правильное предложение языка, порождаемого некоторой грамматикой.

Наоборот, если для некоторой цепочки символов дерево вывода построить невозможно, это значит, что цепочка не принадлежит языку, порождаемому грамматикой.

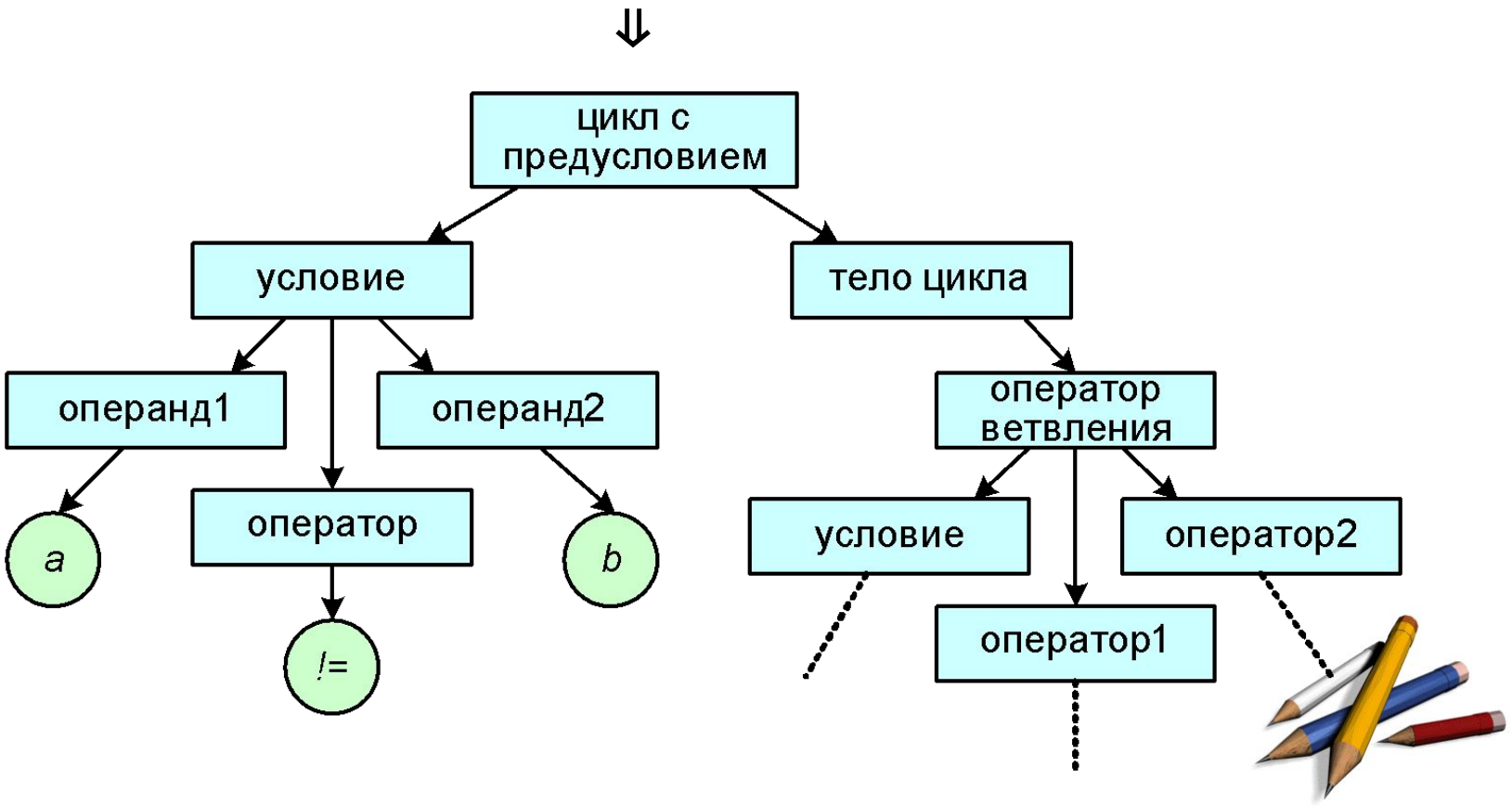


Свойства синтаксического дерева

- Корень дерева помечен стартовым символом грамматики.
- Каждый лист дерева помечен терминальным символом или ε.
- Каждый внутренний узел помечен нетерминальным символом.
- Если нетерминальный символ *A* помечает некоторый внутренний узел, и его дочерние узлы имеют метки $X_1, X_2, ..., X_n$, то грамматика содержит правило вида $A \rightarrow X_1X_2...X_n$, где $X_1, X_2, ..., X_n$ могут быть как терминальными, так и нетерминальными символами.

Пример. Фрагмент синтаксического дерева

while (a!=b) if (a>b) a-=b; else b-=a;

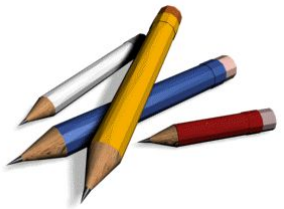


Методы синтаксического анализа

Большинство методов синтаксического анализа делится на два типа:

– нисходящие (сверху вниз);
– восходящие (снизу вверх).

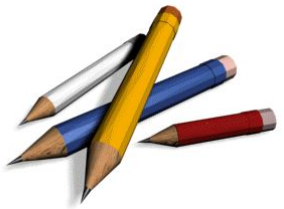
Это связано с порядком, в котором строятся узлы дерева разбора.



Особенности нисходящих методов разбора

– Построение синтаксического дерева начинается от корня по направлению к листьям.

– Удобны для "ручного" программирования синтаксического анализатора.

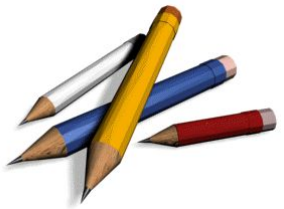


Особенности восходящих методов разбора

– Построение синтаксического дерева начинается от листьев по направлению к дереву.

– Могут быть использованы для более широкого класса грамматик и схем трансляции.

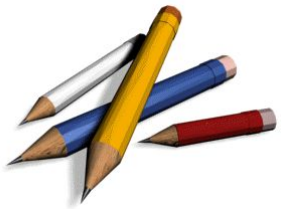
– Обычно используются при автоматизированном построении трансляторов.



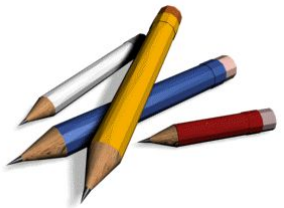
Общий алгоритм методов нисходящего синтаксического анализа

Построение синтаксического дерева начинается с корня, помеченного стартовым нетерминалом, и осуществляется многократным выполнением следующих двух шагов:

1. В узле, помеченном нетерминальным символом *A*, выбираем одно из порождающих правил для *A* и строим дочерние узлы для символов из правой части правила.
2. Находим следующий узел, в котором должно быть построено поддерево.

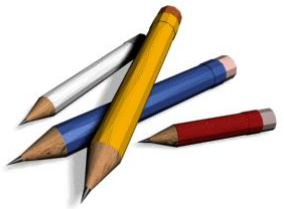


Основные отличия между методами нисходящего анализа заключаются в том, каким образом происходит выбор порождающего правила и как происходит переход к следующему узлу.



Метод рекурсивного спуска

Анализ методом рекурсивного спуска представляет собой способ нисходящего синтаксического анализа, при котором выполняется ряд рекурсивных процедур для обработки входного потока лексем.

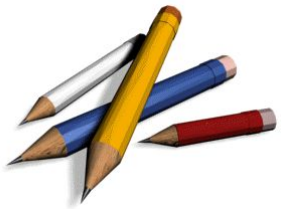


Синтаксический анализ

При выборе порождающего правила для нетерминального символа применяется метод проб и ошибок:

1. делается попытка применить порождающее правило;
2. в случае неуспешной попытки выполняется откат и, затем, переход, к другим порождающим правилам для данного нетерминального символа.

Попытка использования порождающего правила считается неуспешной, если после нее невозможно завершить дерево, соответствующее входной последовательности лексем.



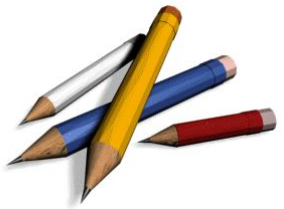
Синтаксический анализ

Для каждого нетерминального символа грамматики записывается отдельная распознающая процедура.

При этом анализ входной последовательности лексем осуществляется по следующему алгоритму:

1. Перед началом работы процедуры текущей является первая лексема анализируемой конструкции языка.

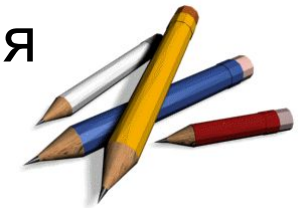
2. В процессе работы распознающая процедура считывает все лексемы, относящиеся к данной конструкции.



2а)Если правило содержит терминальный символ, то процедура должна убедиться в правильности очередной лексемы.

2б)Если правило содержит нетерминальный символ, то процедура должна обратиться к соответствующим распознающим процедурам для анализа части входной последовательности.

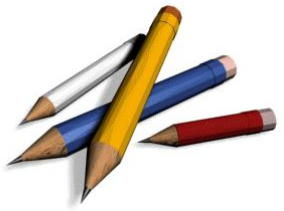
3. В случае успешного окончания работы процедуры текущей становится первая лексема, следующая во входной последовательности за данной конструкцией языка.
4. В случае ошибочного завершения работы процедуры анализа, текущей по прежнему является исходная лексема.



Пример. Шаблон процедуры синтаксического анализа для нетерминального символа "Оператор"

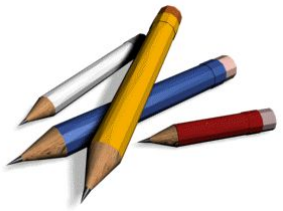
```
boolean Оператор(Узел) {
    // попытки применить возможные порождающие правила
    if ( ПростойОператор(Узел) ) return true;
    if ( СоставнойОператор(Узел) ) return true;
    if ( УсловныйОператор(Узел) ) return true;
    if ( ОператорЦиклаСПредусловием(Узел) ) return true;
    if ( ОператорЦиклаСПостусловием(Узел) ) return true;
    if ( ОператорЦиклаСПараметром(Узел) ) return true;
    . . .
    // ошибочное завершение
    return false;
}
```

Параметр "узел" указывает на текущую вершину, используемую для построения синтаксического дерева



Пример. Шаблон процедуры синтаксического анализа для нетерминального символа "Условный оператор"

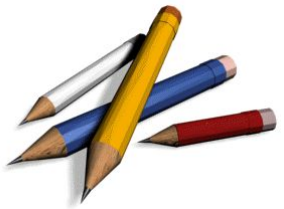
```
boolean УсловныйОператор(Узел) {
    Позиция = СписокЛексем.ТекущаяПозиция;

    ТекущийУзел = new УзелУсловныйОператор;
    if ( СписокЛексем.СледующаяЛексема == ЛЕКСЕМА_IF )
    if ( СписокЛексем.СледующаяЛексема == ЛЕКСЕМА_( )
   if ( Условие(ТекущийУзел.Условие) )
    if ( СписокЛексем.СледующаяЛексема == ЛЕКСЕМА_) )
   if ( Оператор(ТекущийУзел.Оператор1) ) {
        ОператорElse(ТекущийУзел.Оператор2);
        return true;
    }
    // ошибочное завершение
    Узел = Пусто;
    СписокЛексем.ТекущаяПозиция = Позиция;
    return false;
}
```



Предиктивный анализ

Для некоторых грамматик языков программирования допускается построение *детерминированных синтаксических анализаторов*, использующих метод предиктивного (предсказывающего) анализа.

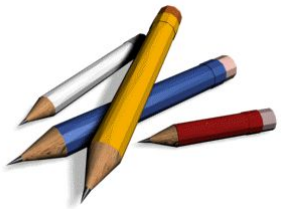
Основное отличие метода предиктивного анализа от метода рекурсивного спуска заключается в том, что на основе анализа нескольких стартовых лексем входной последовательности можно однозначно выбрать порождающее правило грамматики.



Метод предиктивного анализа допустимо применять для LL(k)-грамматик.

LL(k)-грамматикой называется контекстно-свободная грамматика (тип 2), в которой выбор правила в ходе левостороннего вывода однозначно определяется не более чем k очередными символами входной цепочки, считываемой слева направо.

Самым удобным для реализации оказываются синтаксические анализаторы, основанные на LL(1)-грамматиках.

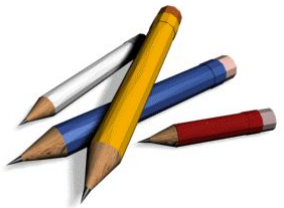


Рассмотрим правило вида A→α.

Определение 1.

Множеством FIRST(α) будем называть множество терминальных символов, которые могут появиться в качестве первого символа последовательностей, полученных из α.

Если α представляет собой ε или может порождать ε, то ε также принадлежит FIRST(α).

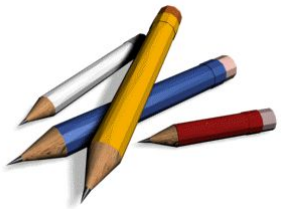


Определение 2.

Множеством FOLLOW(A) будем называть множество терминальных символов, которые могут появиться в непосредственно справа за A в некоторой сентенциальной форме грамматики, т.е. a∈FOLLOW(A), если существует вывод вида

$$S \rightarrow \delta A a \gamma$$

Если A может являться крайним правым символом некоторой сентенциальной формы, то ε также принадлежит FOLLOW(A).



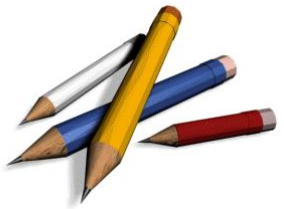
Критерий принадлежности грамматики к классу LL(1)-грамматик

Контекстно-свободная грамматика принадлежит к классу LL(1)-грамматик тогда и только тогда, когда для любых правил вида $A \to \alpha$, $A \to \beta$, принадлежащих грамматике, множества FIRST(α) и FIRST(β) не имеют общих элементов:

$$FIRST(\alpha) \cap FIRST(\beta) = \varnothing$$

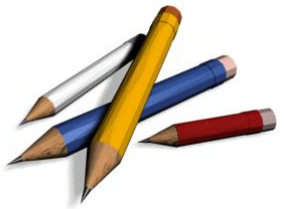
и если $\varepsilon \in FIRST(\alpha)$, то

$$FOLLOW(A) \cap FIRST(\beta) = \varnothing$$



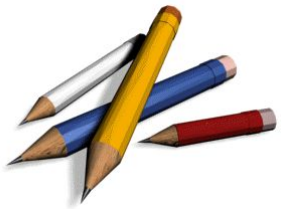
Структура предиктивного анализатора

Предиктивный анализатор содержит набор процедур, соответствующих каждому анализируемому нетерминальному символу. Каждая такая процедура решает две задачи.



1. *Исходя из текущей лексемы принимает решение, какое порождающее правило должно быть использовано.*

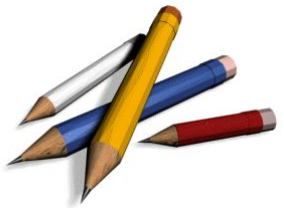
Если сканируемый символ принадлежит множеству FIRST(α), применяется правило с правой частью α. Правило с ε в правой части используется в случае, когда текущий сканируемый символ не принадлежит никакому множеству FIRST(α) для всех возможных правых частей.



2. *Имитирует правую часть порождающего правила.*

Каждый нетерминальный символ, содержащийся в правиле, заменяется на вызов процедуры, соответствующей этому нетерминалу.

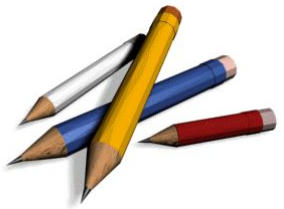
Каждый терминальный символ, содержащийся в правиле, приводит к чтению следующей лексемы из входного потока. Если прочитанная лексема не соответствует терминальному символу, то вызывается процедура обработки ошибки.



Пример. Синтаксический анализ арифметических выражений

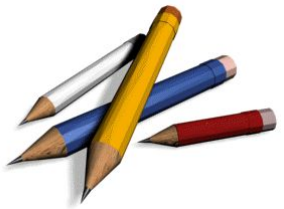
Пусть грамматика арифметических выражений описывается следующими правилами:

expr → term moreterms
moreterms → '–' expr | '+' expr | ε
term → const | ident



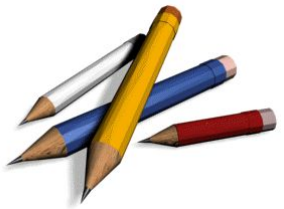
Процедура анализа нетерминального символа "expr"

```
void expr() {
    term(); moreterms();
}
```



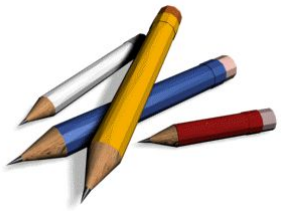
Процедура анализа нетерминального символа "term"

```
void term() {
   if (currentLexem.type==CONST) {
      checkLexem(CONST);
   } else
   if (currentLexem.type==IDENT) {
      checkLexem(IDENT);
   } else {
      error();
   }
}
```



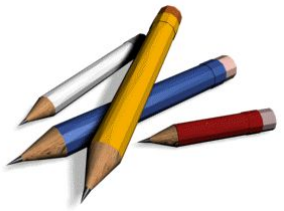
Процедура анализа нетерминального символа "moreterm"

```
void moreterms() {
   if (currentLexem.type==OP_ADD) {
      checkLexem(OP_ADD); expr();
   } else
   if (currentLexem.type==OP_SUB) {
      checkLexem(OP_SUB); expr();
   } else {
      /* допускается пустая цепочка */
   }
}
```



Процедура проверки типа терминального символа

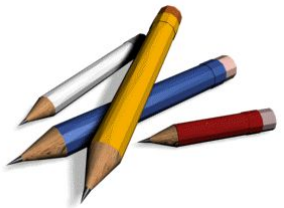
```
void checkLexem(int type) {
   if (currentLexem.type==type) {
      currentLexem=nextLexem();
   } else {
      error();
   }
}
```



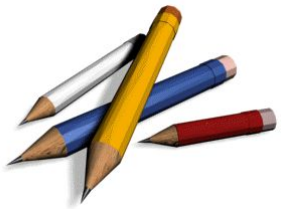
Восходящий синтаксический анализ

Одним из распространенных алгоритмов восходящего синтаксического анализа является алгоритм «сдвиг – свертка».

Основная идея алгоритма заключается в том, что процедура-распознаватель просматривает входную последовательность лексем слева направо и по возможности заменяет некоторую цепочку символов (как терминальных, так и нетерминальных) новым *нетерминальным* символом в соответствии с порождающими правилами грамматики.

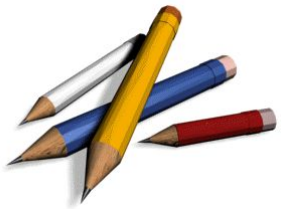


Процедура замены цепочки символов в соответствии с порождающим правилом носит название «*свертка*», процедура считывания очередной лексемы – «*сдвиг*» («*перенос*»).

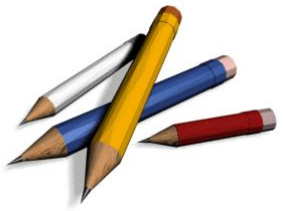


Пример. Анализ выражения **b*b–4*a*c** в соответствии с грамматикой арифметических выражений:

E → E + F | E – F | F * T | F / T | (E) | Ident | Const
F → F * T | F / T | (E) | Ident | Const
T → (E) | Ident | Const

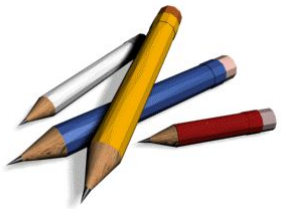


сдвиг	b	Ident
свертка (2)		F
сдвиг	b *	F *
сдвиг	b * b	F * Ident
свертка (3)		F * T
свертка (1)		E
сдвиг	b * b –	E –
сдвиг	b * b – 4	E – Const
свертка (2)		E – F
сдвиг	b * b – 4 *	E – F *
сдвиг	b * b – 4 * a	E – F * Ident



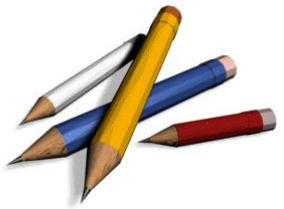
Синтаксический анализ

свертка (3)		E – F * T
свертка (2)		E – F
сдвиг	b * b – 4 * a *	E – F *
сдвиг	b * b – 4 * a * c	E – F * Ident
свертка (3)		E – F * T
свертка (2)		E – F
свертка (1)		E

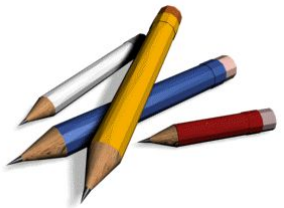


На каждом шаге работы распознаватель должен решать следующие задачи:

1. Какую процедуру необходимо выполнить: сдвиг или свертку?
2. Если выполнять свертку, то какую цепочку выбрать для поиска правил? (какой длины?)
3. Если существует несколько правил с одинаковой правой частью, то какое из них выбрать?



Поскольку каждая из этих задач имеет неоднозначное решение, то при реализации алгоритма на каждом шаге запоминаются все предпринятые действия, для того чтобы иметь возможность вернуться назад и выполнить все действия по другому. Этот процесс повторяется до тех пор, пока не будут перебраны все возможные варианты.



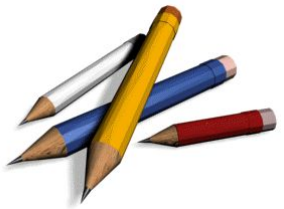
Описание алгоритма «сдвиг–свертка»

Входные данные:

α = $a_1\ a_2\ \ldots\ a_i\ \ldots\ a_N$ – входная цепочка терминальных символов (лексем)

N – количество лексем

P_j, j = 1..M – порождающие правила

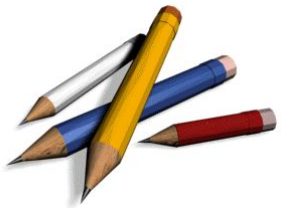


Внутренние переменные

idx – индекс текущей лексемы правила

стек1 – список, который используется для хранения считанных лексем и заменяющих их нетерминальных символов.

стек2 – список, который используется для запоминания примененных правил.

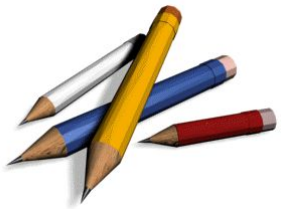


Начальное состояние

idx = 0

стек1 – пустой

стек2 – пустой



Функция 1 (сдвиг – возврат)

Если **idx==N**,

Если **стек1** содержит единственный символ (стартовый символ грамматики),

То успешный выход из функции.

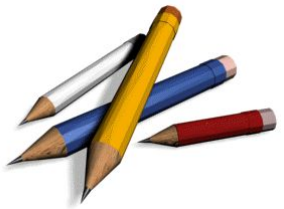
Иначе прекратить алгоритм, сгенерировать ошибку.

Выполнить сдвиг (**idx++** , поместить в **стек1** лексему $\mathbf{a_{idx}}$).

Вызвать функцию 2. Если успешно, то успешный выход из функции.

Реализовать возврат сдвига (извлечь из **стека1** лексему, **idx** – –)

Выход из функции – неуспешно.



Функция 2 (свертка – возврат)

если **idx==N** и **стек1** содержит единственную лексему (стартовый символ), то успешный выход из функции.

Нц. Цикл по порождающим правилам $\mathbf{P_j}$.

Проверить, можно ли выполнить свертку по правилу $\mathbf{P_j}$.

Если можно:

Выполнить свертку (извлечь нужное количество символов из **стека1**, занести в **стек1** новый нетерминал, в **стек2** занести правило $\mathbf{P_j}$).

Вызвать функцию 2. Если успешно, то успешный выход из функции.

Реализовать возврат свертки (извлечь правило из **стека2**, извлечь из **стека1** нетерминал, вернуть в **стек1** правую часть правила).

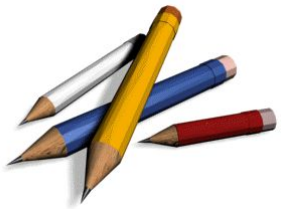
Кц

Вызывать функцию 1. Если успешно, то успешный выход из функции.

Выход из функции – неуспешный

Конечное состояние

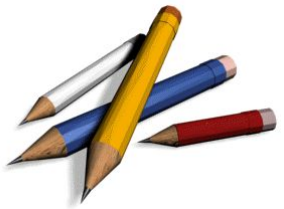
В случае успешного завершения работы алгоритма *стек2* – содержит последовательность правил грамматики, необходимых для построения синтаксического дерева анализируемой строки.



Область применимости алгоритмов типа «сдвиг–свертка»

Исходная грамматика

- не должна содержать циклов (правил вида A→A);
- не должна содержать ε-правил (правил вида A→ε);
- желательно отсутствие цепных правил (правил вида A→B, где A и B – нетерминальные символы).



Пример. Исключение из грамматики цепных правил

Исходная грамматика арифметических выражений

E → E + F | E – F | F
F → F * T | F / T | T
T → (E) | Ident | Const

Преобразованная грамматика

E → E + F | E – F | F * T | F / T | (E) | Ident | Const
F → F * T | F / T | (E) | Ident | Const
T → (E) | Ident | Const