Курс «Компиляторные технологии»

Лекция 5

Синтаксически управляемая трансляция (2)

Курносов Михаил Георгиевич

www.mkurnosov.net

Сибирский государственный университет телекоммуникаций и информатики
Весенний семестр

Упражнения

- **Упражнение 2.2.1.** Рассмотрим контекстно-свободную грамматику

```
S –> S S + | S S * | a
```

- Покажите, как данная грамматика генерирует строку «aa+a*»
- Постройте дерево разбора для данной строки
- Какой язык генерирует данная грамматика?

- **Упражнение 2.2.2.** Какой язык генерируется каждой из следующих грамматик?
  - `S –> 0 S 1 | 0 1`
  - `S –> + S S | - S S | a`
  - `S –> S (S) S | ε`

Трансляция инфиксных выражений в постфиксные

- **Инфиксная запись (infix notation)**
  - a + b — знак операции между операндами
  - a * (b + c) —скобки позволяют задать приоритет подвыражениям
- **Постфиксная запись (postfix notation)**
  - a b + — знак операции после операндов

- **Индуктивное определение постфиксной записи**

1. Если *E* является переменной или константой, то постфиксная запись *E* представляет собой само *E*
2. Если *E* — выражение вида *E*1 **op** *E*2, где **op** — знак бинарной операции, то постфиксная запись *E* представляет собой *E*1 *E*2 **op**, где *E*1 и *E*2 — постфиксные записи для *E*1 и *E*2 соответственно
3. Если *E* — выражение в скобках вида (*E*1), то постфиксная запись для E такова же, как и постфиксная запись для *E*1

- Инфиксная форма: (9—5) + 2
- Постфиксная форма: 95—2+
  9, 5, 2 => 9, 5, 2 (правило 1), 9—5 => 95— (правило 2), (9-5) => 95— (правило 3), (правило 2)

Трансляция инфиксных выражений в постфиксные

- **Скобки в постфиксной записи не используются**, последовательность и арность (arity) — количество аргументов — операторов допускают только единственный способ декодирования постфиксного выражения
- **Вычисление выражения в постфиксной записи**
  1. Выражение в постфиксной записи сканируется слева направо, пока не встретится оператор
  2. Выполняется поиск слева соответствующего количества операндов и найденный оператор выполняется с этими операндами
  3. Результат выполнения замещает операнды и оператор, после чего процесс поиска слева направо очередного оператора продолжается

- Вычисление выражения в постфиксной форме:

9 5 2 + — 3 *

1. Встретили +, применили к двум предшествующим операндам 5+2, заместили 52+ значением 7 => 97—3*
2. Встретили —, применили к двум предшествующим операндам 9—7, заменили значением 2 => 23*
3. Встретили *, применили к двум предшествующим операндам 2*3, заменили значением 6

Синтаксически управляемая трансляция

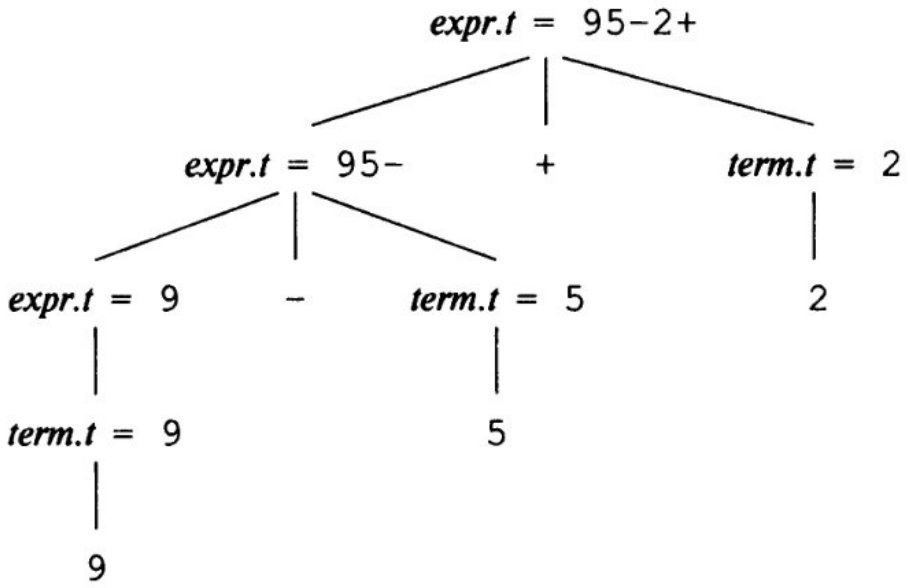
- **Синтаксически управляемая трансляция** (syntax-directed translation) выполняется путем присоединения правил (программных фрагментов) к продукциям грамматики

```
expr -> expr1 + term    {
                          Трансляция expr1
                          Трансляция term
                          Обработка +
                        }
```

- Фрагменты выполняются при использовании продукции в процессе синтаксического анализа
- Объединенный результат выполнения всех фрагментов грамматики в порядке, определяемом синтаксическим анализом, и есть трансляция заданной программы, к которой применяется этот процесс

Синтезированные атрибуты

- Терминалам и нетерминалам грамматики можно назначить атрибуты
- Правила в грамматике, фрагменты кода в продукциях, задают вычисление значений атрибутов в узлах дерева разбора
- **Аннотированное дерево разбора** (annotated parse tree) — дерево разбора с указанием значений атрибутов в каждом узле
- Атрибут называется **синтезированным** (synthesized), если его значение в узле дерева разбора *N* определяется на основании атрибутов дочерних по отношению к *N* узлов и самого узла *N* (вычисление путем единственного восходящего прохода по дереву разбора)

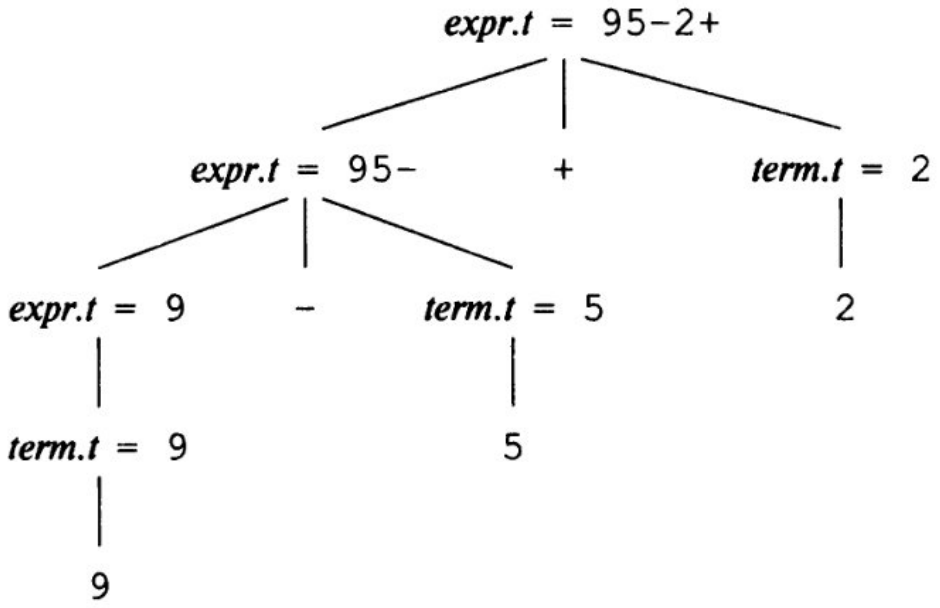


- Синтезированные атрибуты могут быть вычислены при любом восходящем обходе дерева (bottom-up) — обходе, который вычисляет атрибуты узла после вычисления атрибутов дочерних узлов

Аннотированное дерево разбора: значение атрибута *t* в узлах дерева разбора для выражения 95—2+

Семантические правила (semantic rules)

- Назначаем терминалам и нетерминалам аттрибуты (исходя из целей трансляции)
- Назначаем каждой продукции **семантическое правило** (semantic rule) — правило вычисления значений аттрибутов, связанных с символами продукции



Аннотированное дерево разбора 95−2+

Продукция	Семантическое правило
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Синтаксически управляемые определения для трансляции инфиксных выражений в постфиксные (a || b — конкатенация строк a и b)

Простые синтаксически управляемые определения

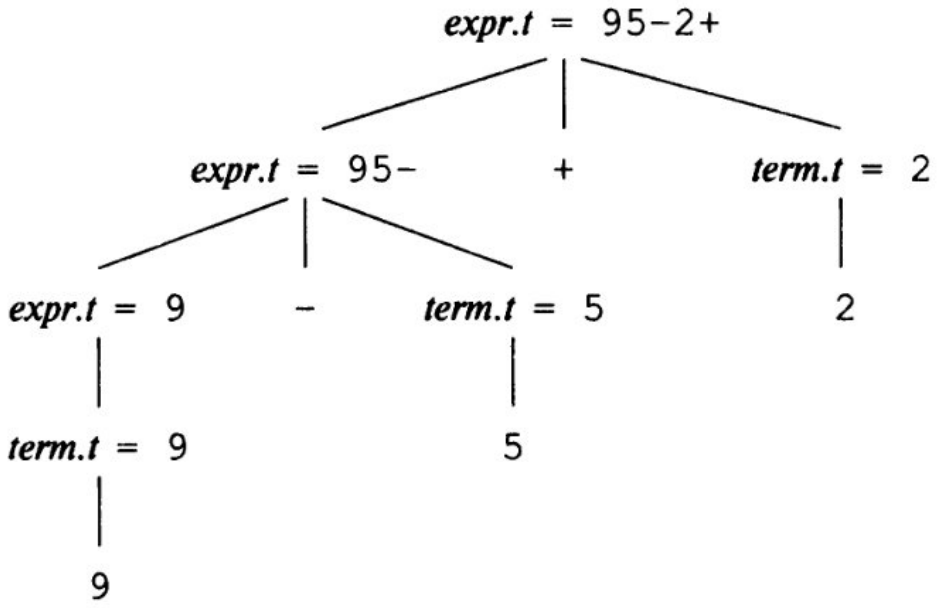
- **Синтаксически управляемое определение называемся простым** (simple), если строка, представляющая трансляцию нетерминала в заголовке каждой продукции, является *конкатенацией трансляций нетерминалов в теле продукции в том же порядке, в котором они встречаются в продукции*, с необязательными дополнительными строками
- Простое синтаксически управляемое определение может быть реализовано путем печати (выдачи) только дополнительных строк в порядке их появления в определении

Продукция	Семантическое правило
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Синтаксически управляемые определения для трансляции инфиксных выражений в постфиксные
(a || b — конкатенация строк a и b)

Семантические правила (semantic rules)

- Назначаем терминалам и нетерминалам аттрибуты (исходя из целей трансляции)
- Назначаем каждой продукции **семантическое правило** (semantic rule) — правило вычисления значений аттрибутов, связанных с символами продукции



Аннотированное дерево разбора 95−2+

Продукция	Семантическое правило
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Синтаксически управляемые определения для трансляции инфиксных выражений в постфиксные (a || b — конкатенация строк a и b)

Семантические правила — операции со строками!
Можно обобщить на программные фрагменты

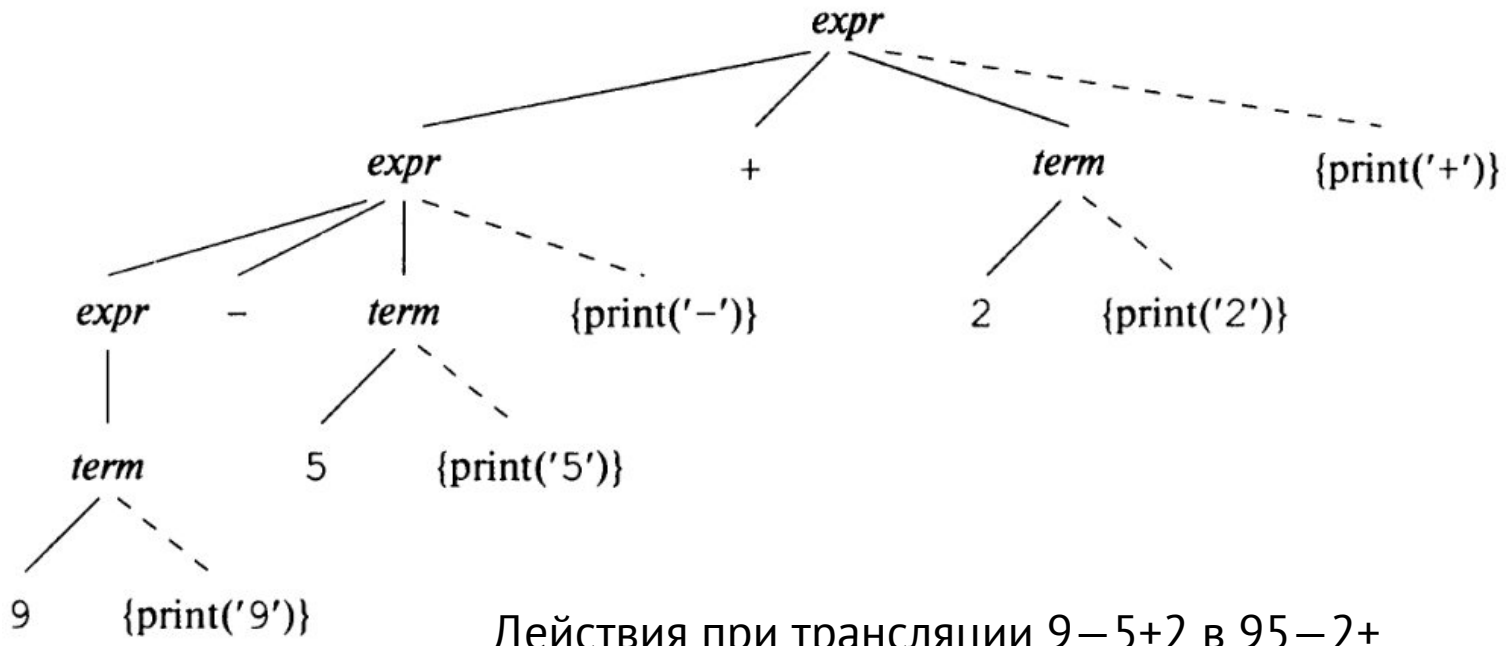
Схемы трансляции (translation schemes)

- **Схема синтаксически управляемой трансляции** — запись для определения конкретной трансляции путем присоединения программных фрагментов (семантических действий) к продукциям грамматики

```
rest –> + term { print('+'); } rest1
```

- В дереве разбора *семантическое действие* (semantic action) указывается добавлением дочернего узла и проведения от него пунктирной линии к узлу, соответствующему заголовку продукции

expr	→	$expr_1$ + *term*	{print('+')}
expr	→	$expr_1$ − *term*	{print('−')}
expr	→	*term*	
term	→	0	{print('0')}
term	→	1	{print('1')}
		...	
term	→	9	{print('9')}



Действия при трансляции 9−5+2 в 95−2+

Разбор (parsing)

- Для любой контекстно-свободной грамматики существует анализатор, который требует для разбора строки из n терминалов время, не превышающее $O(n^3)$
- Для реальных языков программирования можно разработать грамматику, обрабатываемую существенно быстрее
- Для разбора почти всех встречающихся на практике языков программирования можно построить алгоритм с линейным временем разбора $O(n)$
- Анализаторы языков программирования почти всегда делают один проход входного потока слева направо, заглядывая вперед на один терминал, и по ходу просмотра строят части дерева разбора

Методы разбора
(по порядку построения узлов дерева разбора)

Нисходящие
(сверху вниз, top-down)

- Построение узлов дерева разбора от корня к листьям
- Легко построить вручную (hand-written)

Восходящие
(снизу вверх, bottom-up)

- Построение узлов дерева разбора от листьев к корню
- Применимы для большего класса грамматик
- Применяются в генераторах синтаксических анализаторов

Нисходящий анализ (top-down parsing)

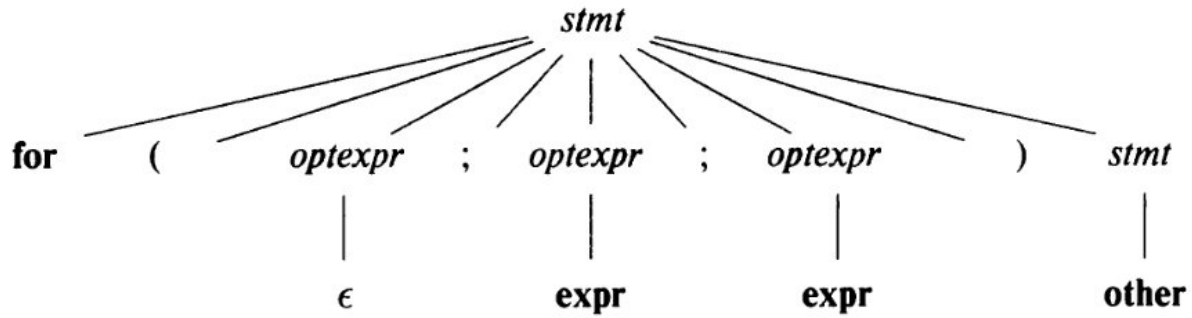
- **Построение дерева разбора методом сверху вниз** (top-down) начинается с корня, стартового нетерминала, и осуществляется многократным выполнением двух шагов:
  1. В узле *N*, помеченном нетерминалом *A* выбираем одну из продукций для *A* и строим дочерние узлы *N* для символов из правой части продукции
  2. Находим следующий узел, в котором должно быть построено поддерево (обычно это крайний слева неразвернутый нетерминал дерева)

Грамматика подмножества языка C

stmt	→	**expr** ;
	|	**if** (**expr**) *stmt*
	|	**for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*
	|	**other**
optexpr	→	ϵ
	|	**expr**

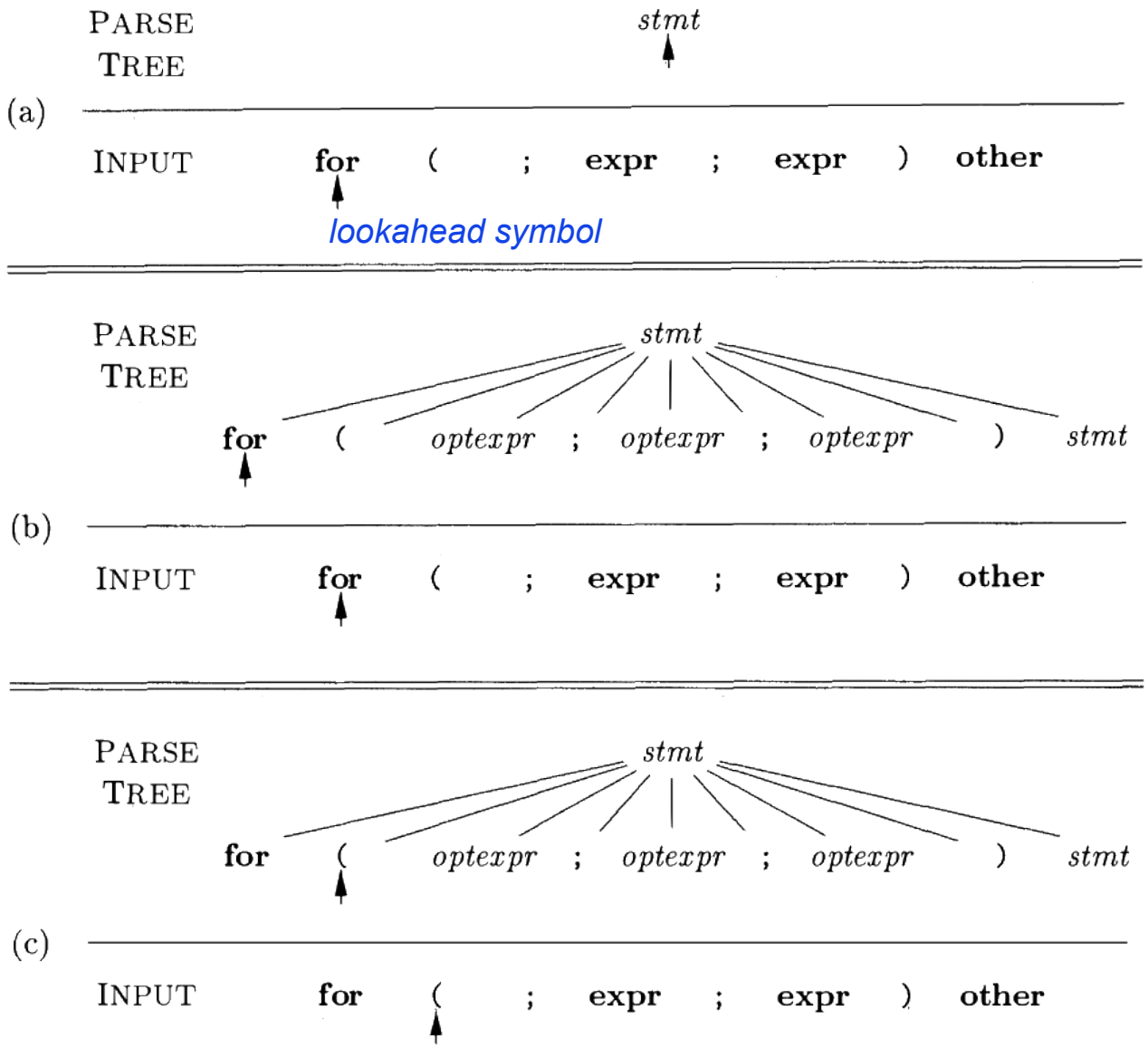
Дерево разбора методом сверху вниз для строки

`for ( ; expr; expr ) other`



Нисходящий анализ (top-down parsing)

- Для некоторых грамматик построение дерева разбора может быть реализовано за один проход слева направо по входной строке
- Текущий сканируемый терминал входной строки называют *сканируемым символом, символом предпросмотра* или "предсимволом" (lookahead symbol)
- Как только у узла дерева разбора создаются дочерние узлы, следует рассмотреть крайний слева узел



Анализ методом рекурсивного спуска (recursive-descent parsing)

- **Анализ методом рекурсивного спуска** (recursive-descent parsing) — тип нисходящего синтаксического анализа, при котором для обработки входной строки используется множество рекурсивных процедур, для каждого нетерминала грамматики
- **Предиктивный анализ методом рекурсивного спуска** (предсказывающий, predictive parsing) — сканируемый символ однозначно определяет поток управления в теле рекурсивной процедуры для каждого нетерминала
- В предиктивном анализе последовательность вызовов процедур при обработке входной строки неявно определяет его дерево разбора и при необходимости может использоваться для явного построения дерева

Предиктивный анализ методом рекурсивного спуска (predictive recursive-descent parsing)

Грамматика подмножества языка C

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{expr ;} \\
 & | & \textbf{if (expr)} \textit{ stmt} \\
 & | & \textbf{for (} \textit{ optexpr ; optexpr ; optexpr } \textbf{)} \textit{ stmt} \\
 & | & \textbf{other} \\
 & & \\
\textit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

- Анализ начинается с вызова процедуры для стартового нетерминала stmt()

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

Предиктивный анализ методом рекурсивного спуска (predictive recursive-descent parsing)

Грамматика подмножества языка C

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{expr ;} \\
 & | & \textbf{if (expr)} \ stmt \\
 & | & \textbf{for (} \ optexpr \ \textbf{;} \ optexpr \ \textbf{;} \ optexpr \ \textbf{)} \ stmt \\
 & | & \textbf{other} \\
 & & \\
optexpr & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

```
void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(';'); optexpr(); match(';'); optexpr();
        match(')'); stmt(); break;
    case other;
        match(other); break;
    default:
        report("syntax error");
    }
}
```

- Предиктивный анализ основан на информации о первых символах, которые могут быть сгенерированы телом продукции
- FIRST(α) — множество терминалов, которые могут появиться в качестве первого символа одной или нескольких строк, сгенерированных из α
- α начинается либо с терминала, который, является единственным символом в FIRST(α), либо с нетерминала
- `FIRST(stmt) = {expr, if, for, other};  FIRST(expr ;) = {expr}`

Предиктивный анализ методом рекурсивного спуска (predictive recursive-descent parsing)

Грамматика подмножества языка C

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{expr ;} \\
 & | & \textbf{if (expr)} \ stmt \\
 & | & \textbf{for (} \ optexpr \ \textbf{;} \ optexpr \ \textbf{;} \ optexpr \ \textbf{)} \ stmt \\
 & | & \textbf{other} \\
 & & \\
optexpr & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

```
void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(';'); optexpr(); match(';'); optexpr();
        match(')'); stmt(); break;
    case other;
        match(other); break;
```

- Если в грамматике присутствуют две продукции:
  - A —> α
  - A —> β
- Предиктивный анализатор требует, чтобы множества FIRST(α) и FIRST(β) были непересекающимися.
- Это обеспечивает возможность использования текущего сканируемого символа (lookahead) для принятия решения, какую из продукций следует применить
- Если сканируемый символ (lookahead) принадлежит множеству FIRST(α), используется продукция для α; в противном случае, если сканируемый символ принадлежит множеству FIRST(β), применяется продукция β

Реализация предиктивного анализатора

1. **Построить предиктивный анализатор, игнорируя действия в продукциях**
2. **Скопировать в анализатор действия из *схемы трансляции*** (продукции + семантические действий)
   - Если действие в продукции *p* находится после символа грамматики *X*, то оно копируется после кода, реализующего *X*.
   - В противном случае, если это действие располагается в начале продукции, оно копируется непосредственно перед кодом, реализующим продукцию

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{expr}\ ; \\
 & | & \textbf{if}\ (\ \textbf{expr}\ )\ stmt \\
 & | & \textbf{for}\ (\ optexpr\ ;\ optexpr\ ;\ optexpr\ )\ stmt \\
 & | & \textbf{other} \\
 & & \\
optexpr & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

Левая рекурсия в продукциях

- **Анализатор на основе рекурсивного спуска зациклиться при "леворекурсивных" продукциях типа:**

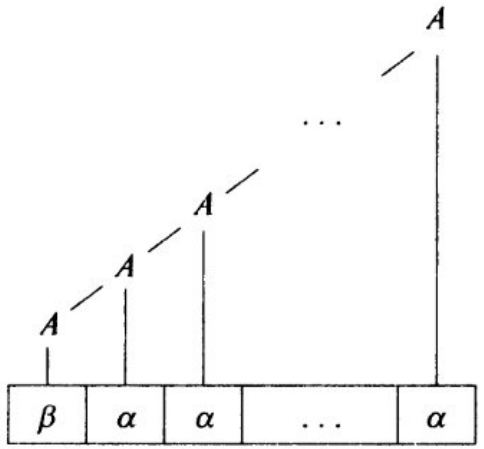
```
expr –> expr + term
```

```
void expr() {
    expr();
    match(‘+’)
    term();
}
```

- Сканируемый символ lookahead изменяется только тогда, когда он соответствует терминалу в теле продукции, между рекурсивными вызовами expr() не происходит никаких изменений
- Как устранить левую рекурсию в продукциях?

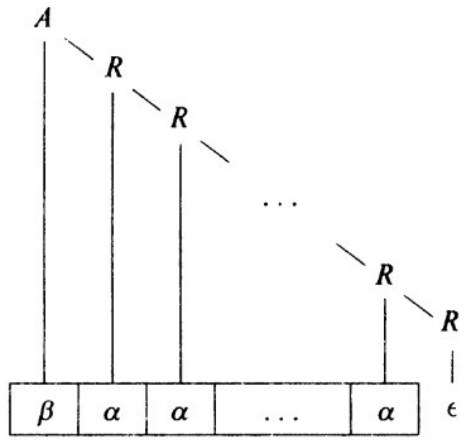
Устранение левой рекурсии в продукциях

- **Продукции с левой рекурсией** — A содержит A в качестве крайнего слева символа в теле продукции
  - A –> Aα | β
  - `expr –> expr + term | term`
- α, β — последовательности терминалов и нетерминалов, которые не начинаются с A
  - `expr –> expr + term | term`



Порождение строки βαααα…α леворекурсивной грамматикой

- **Устранение левой рекурсией введением промежуточного нетерминала (праворекурсивная грамматика)**
  - A –> βR
  - R –> αR | ε
- Рост деревьев вниз вправо затрудняет трансляцию выражений, содержащих левоассоциативные операторы (–, +)

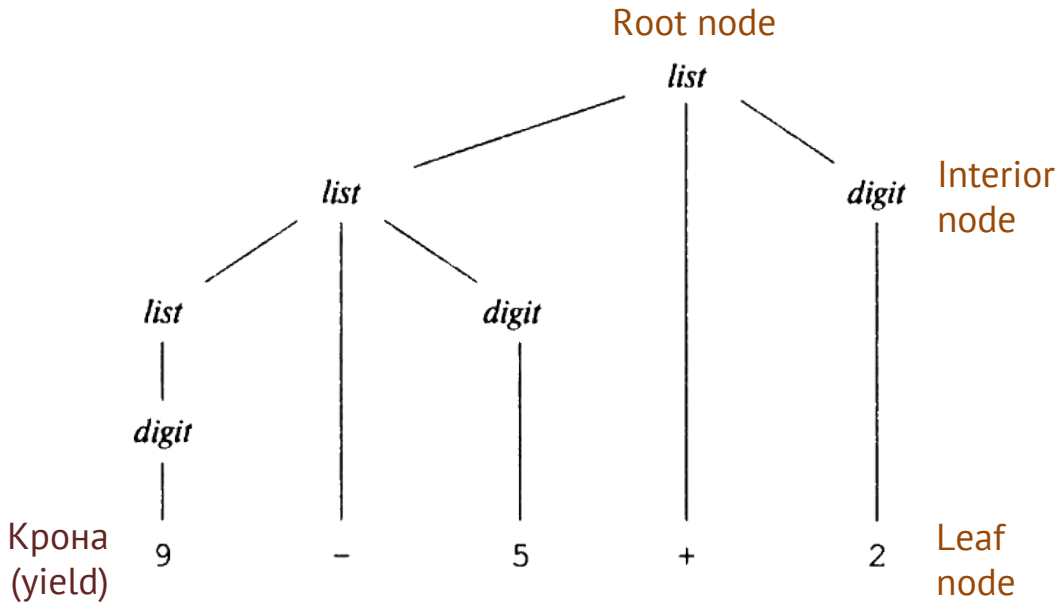


Порождение строки βαααα…α праворекурсивной грамматикой

Абстрактное синтаксическое дерево

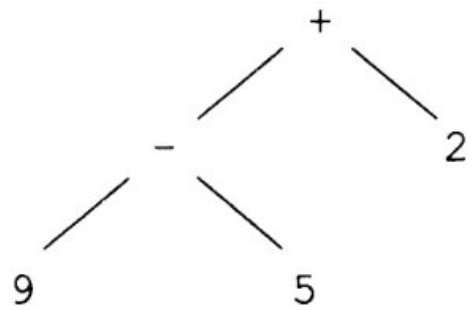
Дерево разбора (parse tree) — древовидное представление порождения строки языка из стартового символа грамматики

Дерево разбора для строки «9-5+2»



Абстрактное синтаксическое дерево (abstract syntax tree) — древовидное представление порождения строки языка, в котором узлами являются программные конструкции

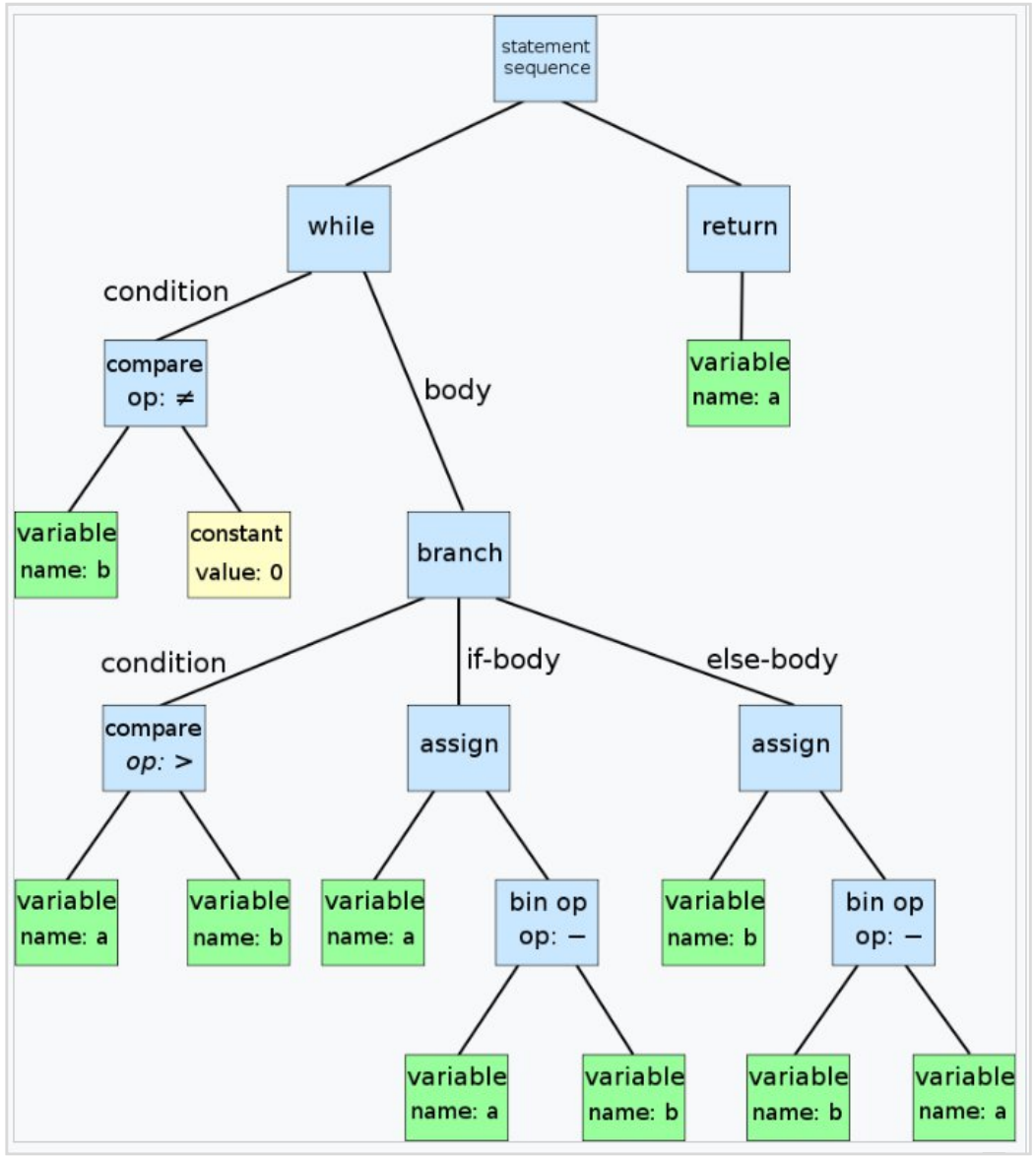
Абстрактное синтаксическое дерево разбора для строки «9-5+2»



Абстрактное синтаксическое дерево

- **Абстрактное синтаксическое дерево** (abstract syntax tree, AST) — конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья — с соответствующими операндами
- Листья являются пустыми операторами и представляют только переменные и константы

```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```



Транслятор простых выражений

- Транслятор арифметических выражений из инфиксной формы в постфиксную запись
- Схема трансляции (продукции + семантические действия)

$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{\ \text{print}('+')\ \} \\
 & | & expr - term & \{\ \text{print}('-')\ \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & 0 & \{\ \text{print}('0')\ \} \\
 & | & 1 & \{\ \text{print}('1')\ \} \\
 & & \ldots & \\
 & | & 9 & \{\ \text{print}('9')\ \}
\end{array}
$$

- Подходит ли грамматика для построения предиктивного синтаксического анализатора методом рекурсивного спуска?

Транслятор простых выражений

- Транслятор арифметических выражений из инфиксной формы в постфиксную запись
- Схема трансляции (продукции + семантические действия)

$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{\ \text{print}('+')\ \} \\
 & | & expr - term & \{\ \text{print}('-')\ \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & 0 & \{\ \text{print}('0')\ \} \\
 & | & 1 & \{\ \text{print}('1')\ \} \\
 & & \ldots & \\
 & | & 9 & \{\ \text{print}('9')\ \}
\end{array}
$$

Левая рекурсия!

Левая рекурсия!

- Подходит ли грамматика для построения предиктивного синтаксического анализатора методом рекурсивного спуска?

Устранение левой рекурсии

- **Метод устранения левой рекурсии** трансформирует продукции вида

  A –> Aα | Aβ | γ

- в продукции:

  A –> γR
  R –> αR | βR | ε

A A α

expr → *expr* + *term* { print('+') }
| *expr* − *term* { print('−') } β
| *term* γ

term → 0 { print('0') }
| 1 { print('1') }
...
| 9 { print('9') }

Устранение левой рекурсии →

R

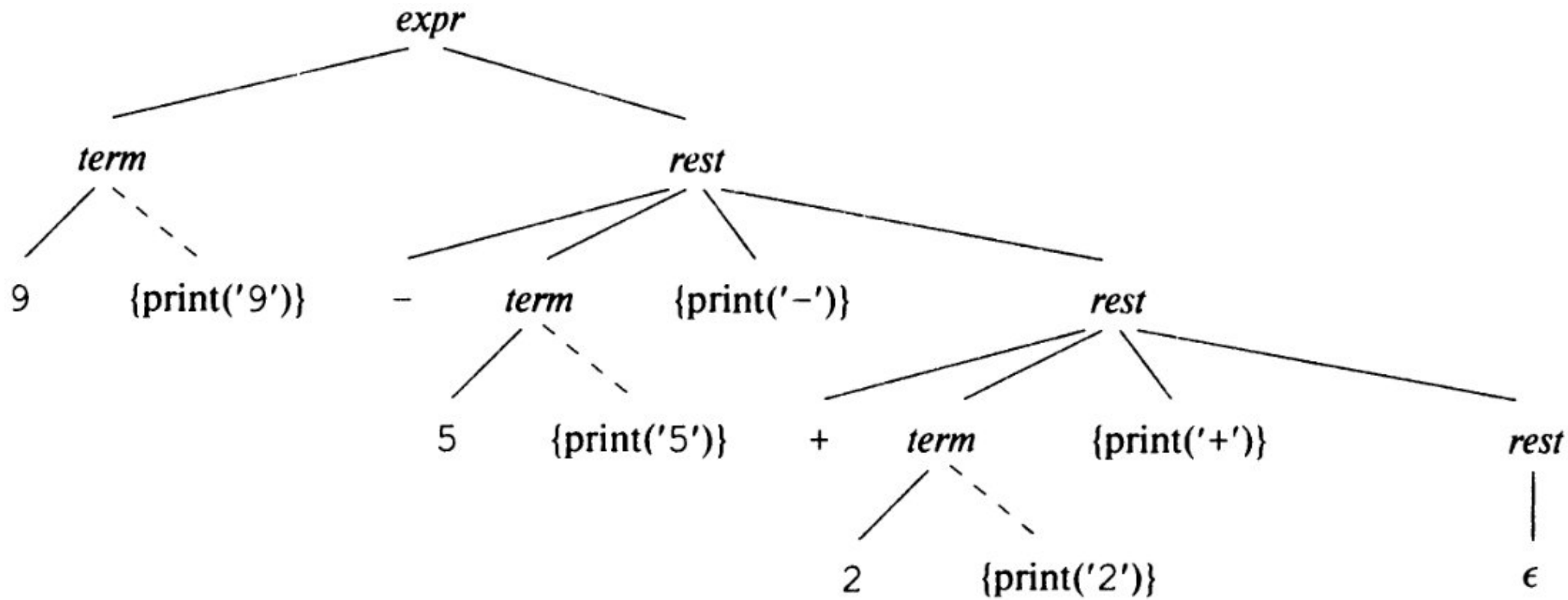
expr → *term rest*

R

rest → + *term* { print('+') } *rest*
| − *term* { print('−') } *rest*
| ϵ

term → 0 { print('0') }
| 1 { print('1') }
...
| 9 { print('9') }

Трансляция 9−5+2 в 95−2+



Предиктивный анализатор рекурсивным спуском

Схема трансляции

$$
\begin{array}{lcl}
expr & \rightarrow & term\ rest \\
rest & \rightarrow & +\ term\ \{\ \text{print}('+')\ \}\ rest \\
 & | & -\ term\ \{\ \text{print}('-')\ \}\ rest \\
 & | & \epsilon \\
term & \rightarrow & 0\ \{\ \text{print}('0')\ \} \\
 & | & 1\ \{\ \text{print}('1')\ \} \\
 & & \ldots \\
 & | & 9\ \{\ \text{print}('9')\ \}
\end{array}
$$

Анализатор рекурсивным спуском

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

void rest() {
        if ( lookahead == '+' ) {
                match('+'); term(); print('+'); rest();
        }
        else if ( lookahead == '-' ) {
                match('-'); term(); print('-'); rest();
        }
        else { } /* Не делать ничего */ ;
}

void term() {
        if ( lookahead — цифра ) {
                t = lookahead; match(lookahead); print(t);
        }
        else report("syntax error");
}
```

Упрощение анализатора: устранение хвостовой рекурсии

Анализатор рекурсивным спуском

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

void rest() {
        if ( lookahead == '+' ) {
                match('+'); term(); print('+'); rest();
        }
        else if ( lookahead == '-' ) {
                match('-'); term(); print('-'); rest();
        }
        else { } /* Не делать ничего */ ;
}

void term() {
        if ( lookahead — цифра ) {
                t = lookahead; match(lookahead); print(t);
        }
        else report("syntax error");
}
```

- **Хвостовая рекурсия** (tail recursion) — рекурсивный вызов той же самой функции, стоящий последней выполняемой инструкцией в её теле
- Может быть легко заменён на итерацию (цикл) путём формальной и гарантированно корректной перестройки кода функции

1. Замена тела rest() на цикл
2. Включение тела rest() в функцию expr(), вместо вызова

```
void rest() {
        while( true ) {
                if( lookahead == '+' ) {
                        match('+'); term(); print('+'); continue;
                }
                else if ( lookahead == '-' ) {
                        match('-'); term(); print('-'); continue;
                }
                break ;
        }
}
```

Предиктивный анализатор рекурсивным спуском

```
class Parser {
public:
    void parse(const std::string& e) {
        ss << e;
        lookahead = readChar();
        expr();
        writeChar('\n');
    }

private:
    std::stringstream ss;
    char lookahead;

    void expr() {
        term();
        while (true) {
            if (lookahead == '+') {
                match('+'); term(); writeChar('+');
            } else if (lookahead == '-') {
                match('-'); term(); writeChar('-');
            } else {
                return;
            }
        }
    }
```

```
void term()
{
    if (std::isdigit(lookahead)) {
        writeChar(lookahead); match(lookahead);
    } else {
        std::cerr <<
            "Syntax error: expected digit\n";
        std::exit(EXIT_FAILURE);
    }
}

void match(char ch)
{
    if (lookahead == ch) {
        lookahead = readChar();
    } else {
        std::cerr <<
            "Syntax error: expected sym '"
            << ch << "'\n";
        std::exit(EXIT_FAILURE);
    }
}
```

Предиктивный анализатор рекурсивным спуском

```
    char readChar()
    {
        char ch;
        ss >> ch;
        return ch;
    }

    void writeChar(char ch) const
    {
        std::cout << ch;
    }
};

int main()
{
    Parser parser;
    parser.parse({"2+5-9"});
    return 0;
}
```

```
$ ./parser
25+9-
```