

Компиляторные технологии

Введение: Общие принципы работы компилятора

Мельник Дмитрий Михайлович

dm@ispras.ru

Материалы курса: compilers.ispras.ru

Вопросы: dm@ispras.ru

Планируется 6 контрольных (4 по машинно-независимым, 2 по машинно-зависимым оптимизациям), по их результатам можно получить «автомат» за экзамен

**Первая лекция – введение и обзор области
(материал этой лекции не войдет в контрольные)**

Литература (1)



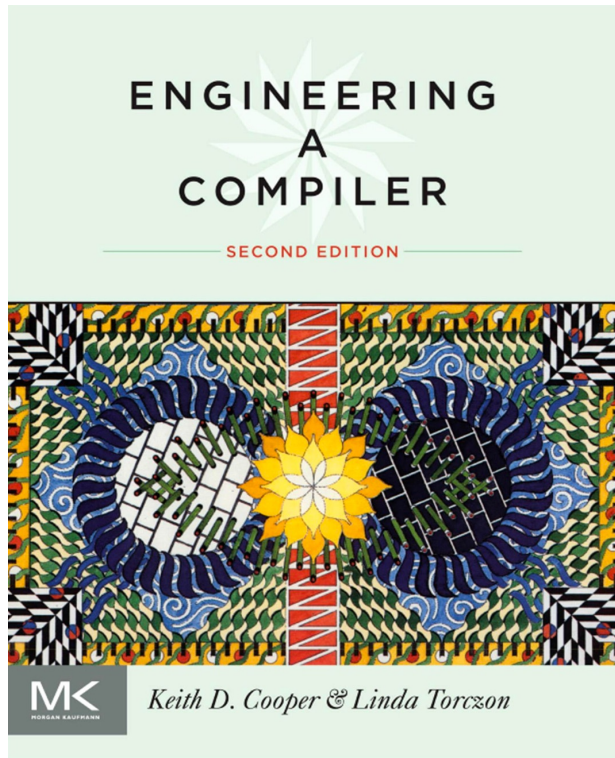
“Dragon Book”

Ахо, Лам, Сети, Ульман.

**Компиляторы. Принципы,
технологии, инструменты.**

2-е издание (2008)

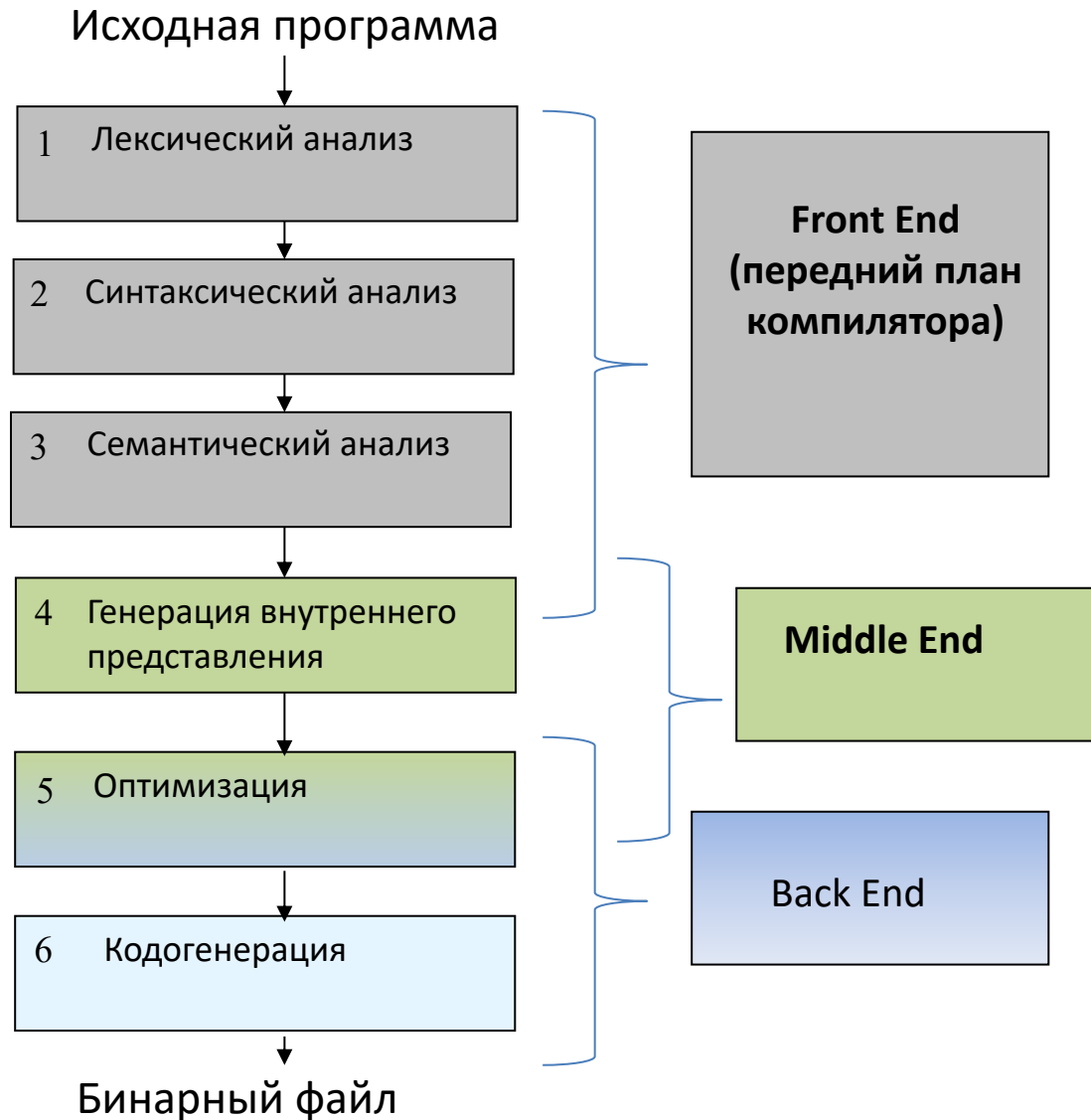
Литература (2)



**Keith D. Cooper, Linda Torczon.
Engineering a Compiler (Second
Edition)**

Elsevier, Inc. 2012

Фазы компиляции



Препроцессирование

- Сформировать входную программу для компилятора
- Макросы – текстовые подстановки

```
#define i j    // Happy debugging!
```

- Включение файлов

```
#include "headers.h"
```

- Скрытие деталей реализации

```
#define tolower(c) __tobody (c, tolower, *__ctype_tolower_loc (), (c))
```

- Выполнить препроцессирование:

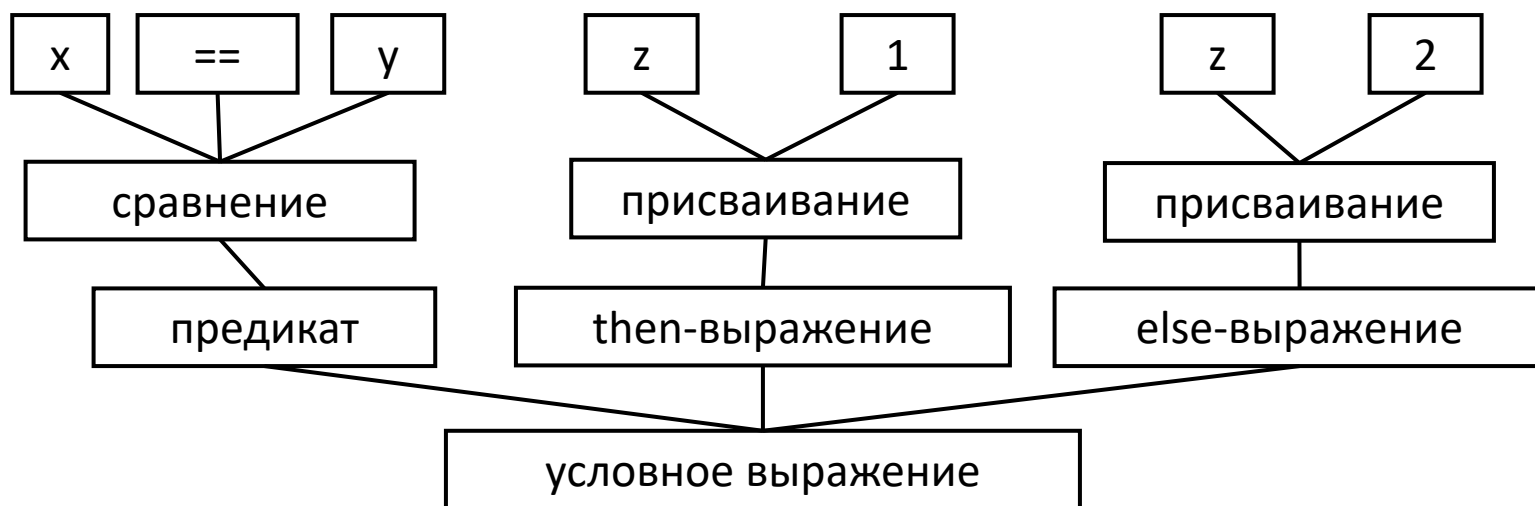
```
gcc test.c -E -o test.i
```

Лексический анализ

- В естественных языках:
 - *Лексический анализ слова – это разбор слова с точки зрения его значения, происхождения, употребления, наличия у него синонимов, антонимов, многозначности.*
- В компиляторе:
 - Цель: разбить текст (программу) на слова (лексемы)
 - Выполняется за один проход по тексту
 - Незначащие символы удаляются
 - Пробелы, комментарии
 - `if x == y then z = 1e10; else z = 3.14;`
 - Синтаксический анализ будет работать с последовательностью **лексем**:
 - *if, x, ==, y, then, z, =, 1e10, ,, else, z, =, 3.14, ;*

Синтаксический анализ

- Выделить предложения и разобрать их структуру по правилам грамматики языка
- Часто требует рекурсивного обхода дерева, заглядывания вперед на несколько шагов
- **if x == y then z = 1; else z = 2;**



Семантический анализ

- Предложения языка могут быть многозначными
- Я встретил её на поляне с цветами
 - Где находились цветы?
- Jack said Jack forgot his textbook at home
 - Сколько всего Джеков?
 - Кто из них забыл учебник?

Семантический анализ

- Грамматика языка дополняется правилами, позволяющими избегать двусмысленных трактовок
- Внутреннее определение переменной `j` перекрывает внешнее
- Необходимы дополнительные проверки этих правил
 - Не больше одной переменной `j` на уровень вложенности
- “Лена забыла дома своя учебник”
 - Несогласованность (несоответствие типов) между “своя” и “учебник”

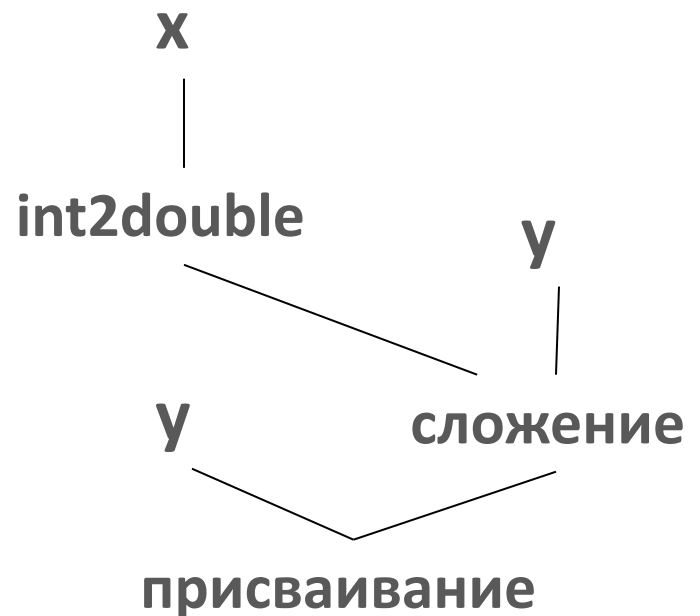
```
{  
    int j = 3;  
    {  
        int j = 4;  
        printf("%d\n", j);  
    }  
}
```

Семантический анализ

- Результирующее дерево может содержать дополнительные операции, вставленные на этом этапе

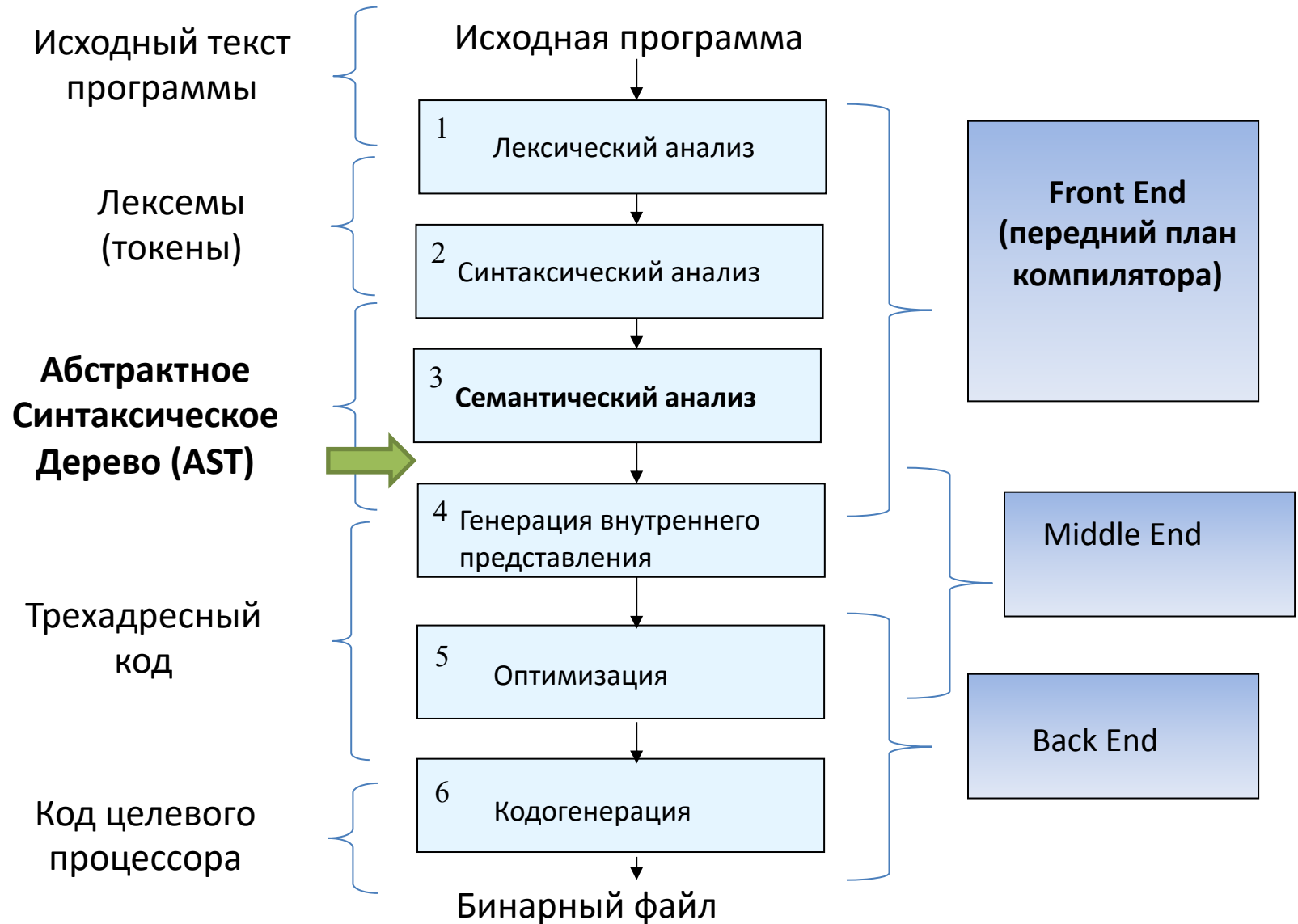
```
int x, double y;
```

```
y = x + y;
```



- Таблица имен заполняется в ходе всего анализа, начиная с лексического

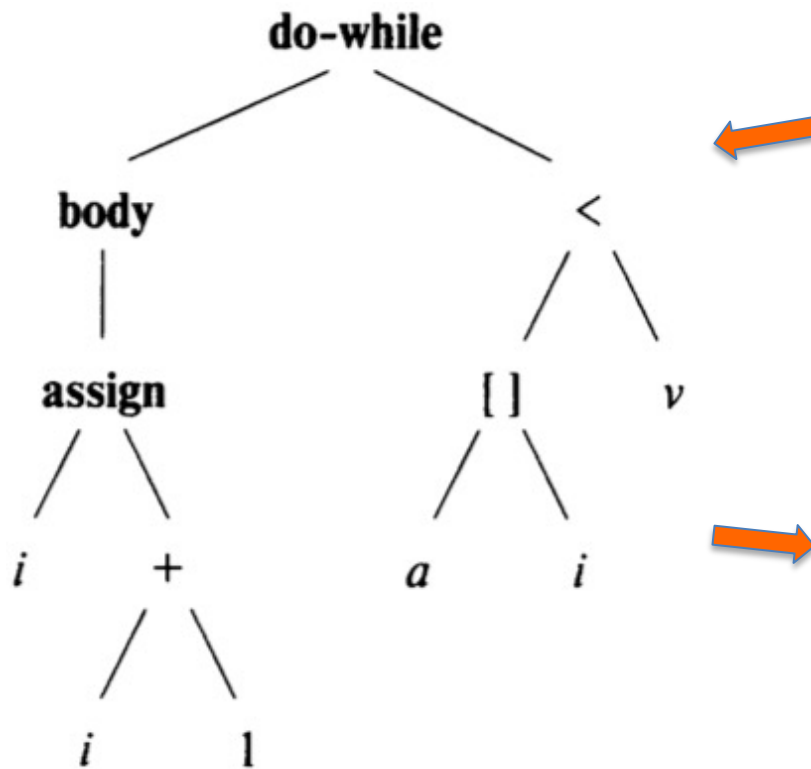
Фазы компиляции



Структура front-end'a



Трансляция во внутреннее представление



Абстрактное Синтаксическое
Дерево (AST)

Исходная программа

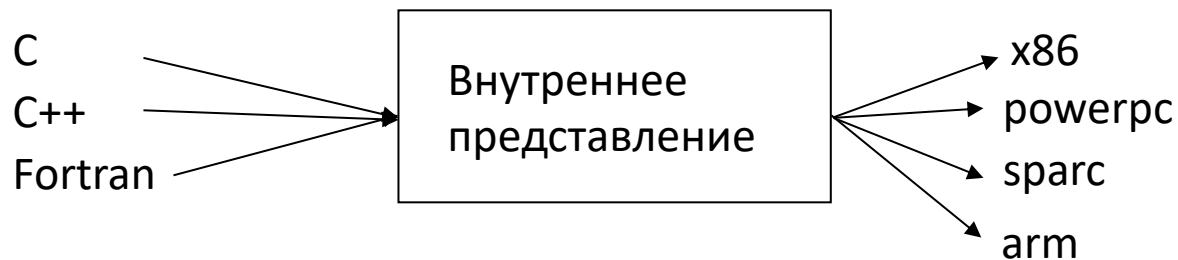
```
do {  
    i = i + 1 ;  
} while (a[i] < v) ;
```

```
1: i = i + 1  
2: t1 = a [ i ]  
3: if t1 < v goto 1
```

Результат трансляции: промежуточное
представление (трехадресный код)

Зачем нужно внутреннее представление

- Удобство выполнения анализа и синтеза
- Возможность построить компилятор с нескольких языков на несколько архитектур (GCC, Clang)
- Для разных этапов – разные представления



Внутреннее представление

- Трехадресное представление:

```
dest = OP arg1, arg2
```

- Неограниченное число псевдорегистров

- LLVM:

```
%3 = load i64, i64* %1  
%4 = load i64, i64* %2  
%5 = add nsw i32 %3, %4
```

- GCC:

```
(set (mem:SI (plus:SI (reg:SI 3 bx) (const_int 4)))  
      (reg:SI 85))
```

Представление для операции записи в память `[bx + 4] = @reg85`

Аппаратные регистры (bx) и адресация памяти ([bx+4]) появляются на более поздних этапах компиляции, когда выполняются машинно-зависимые оптимизации.

Трансляция во внутреннее представление

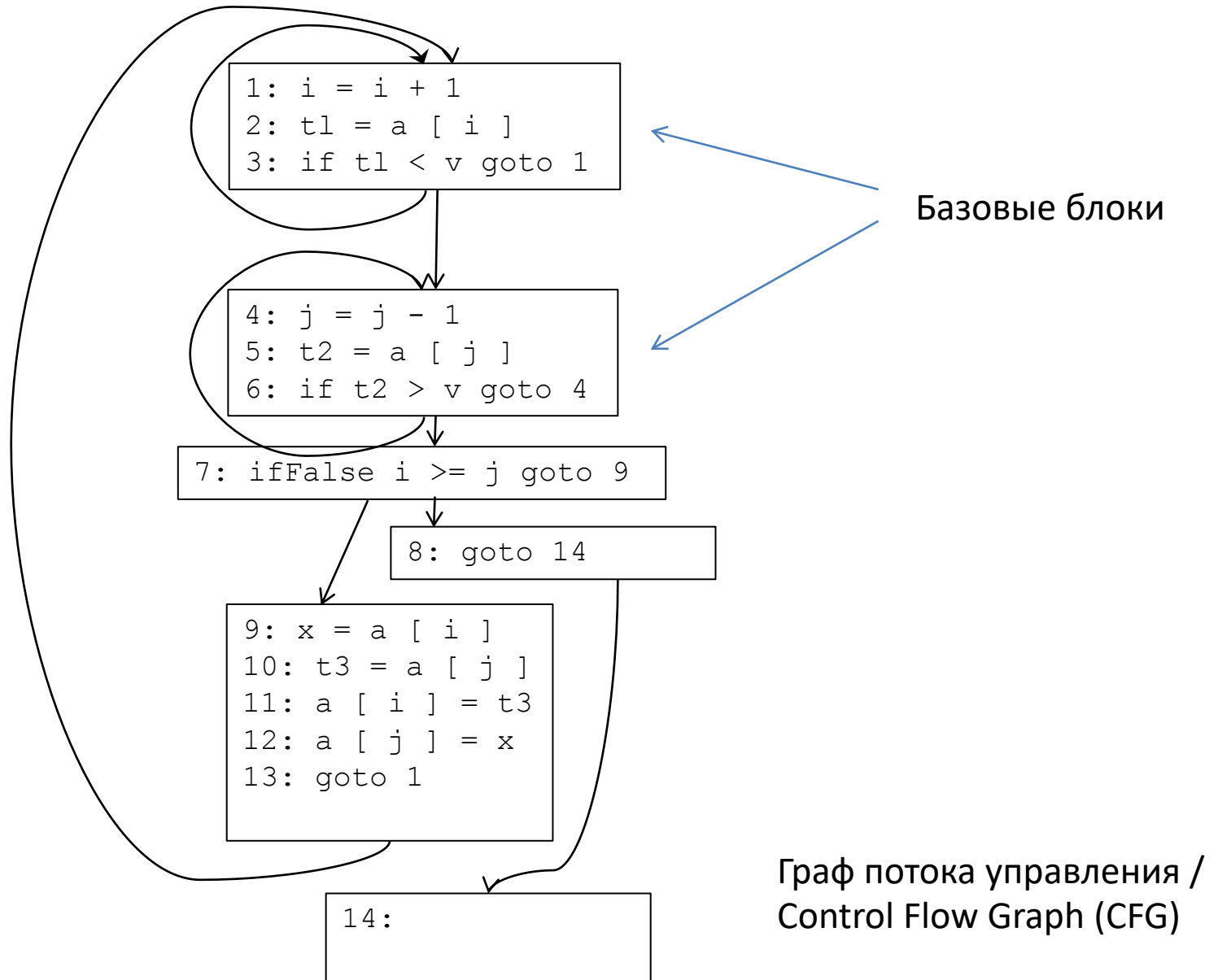
```
{
  int i; int j; float[100] a;
  float v; float x;
  while ( true ) {
    do i = i+1; while ( a[i] < v );
    do j = j-1; while ( a[j] > v );
    if ( i >= j ) break;
    x = a[i];
    a[i] = a[j];
    a[j] = x;
  }
}
```

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
4: j = j - 1
5: t2 = a [ j ]
6: if t2 > v goto 4
7: ifFalse i >= j goto 9
8: goto 14
9: x = a [ i ]
10: t3 = a [ j ]
11: a [ i ] = t3
12: a [ j ] = x
13: goto 1
14:
```

Исходная программа

Результат трансляции: промежуточное представление (трехадресный код)

Граф Поточка Управления

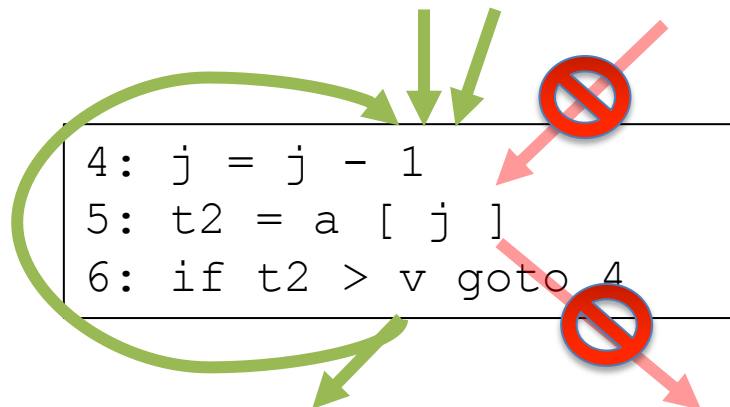


Базовые блоки

Базовый блок (или *линейный участок*) – последовательность следующих одна за другой трехадресных инструкций, обладающая следующими свойствами:

- поток управления может входить в базовый блок только через его первую инструкцию, т.е. в программе нет переходов в середину базового блока;
- поток управления покидает базовый блок без останова или ветвления, за исключением, возможно, в последней инструкции базового блока.

Пример:



Дуги, передающие управление в/из середины базового блока, недопустимы (в таких случаях должны быть созданы отдельные базовые блоки)

Синтаксический Анализ

Грамматики

Пример правила грамматики (*продукция*):

$$stmt \rightarrow \mathbf{if} \ (\ expr \) \ stmt \ \mathbf{else} \ stmt$$

Продукция определяет возможный вид языковой конструкции.

if , (,), **else** – терминалы (*токены*)

stmt, *expr* – нетерминалы

ϵ – специальный терминал (пустая строка)

Грамматика состоит из множества продукций, терминальных и нетерминальных символов и стартового символа.

Грамматики

Пример грамматики:

$list^* \rightarrow list + digit$

$list \rightarrow list - digit$

$list \rightarrow digit$

$digit \rightarrow 0|1|2|3|4|5|6|7|8|9$

Грамматика выводит (порождает) строки, начиная с стартового символа, подставляя вместо нетерминалов правую часть продукций. Строки терминалов, порождаемые грамматикой из стартового символа, образуют *язык*, определяемый грамматикой.

Грамматики

1: $list^* \rightarrow list + digit \mid list - digit \mid digit$

2: $digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Разбор строки (parsing):

9 – 5 + 2

9 $\Rightarrow$ digit (2); 5 $\Rightarrow$ digit (2); digit $\Rightarrow$ list (1c)

list – digit + 2

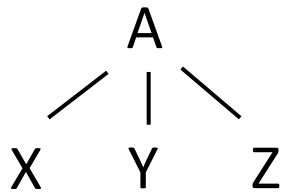
list – digit $\Rightarrow$ list (1b)

list + 2

2 $\Rightarrow$ digit (2); list + digit $\Rightarrow$ list* (1a)

Дерево разбора

$A \rightarrow XYZ$

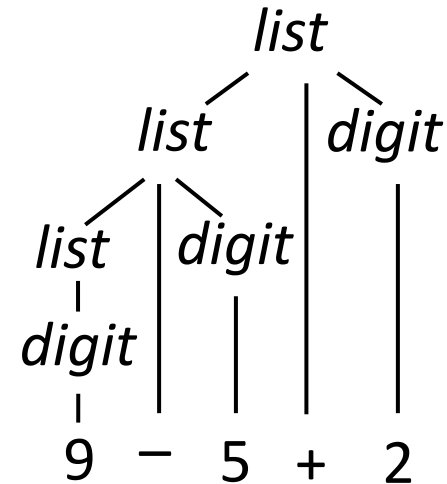


$list^* \rightarrow list + digit$

$list \rightarrow list - digit$

$list \rightarrow digit$

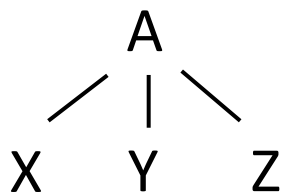
$digit \rightarrow 0|1|2|3|4|5|6|7|8|9$



1. Корень дерева помечен стартовым символом.
2. Каждый лист помечен терминалом или ϵ .
3. Каждый внутренний узел помечен нетерминалом.
4. Если A является нетерминалом и помечает некоторый внутренний узел, а $X_1, X_2, \dots, X_n$ — метки его дочерних узлов слева направо, то должна существовать продукция $A \rightarrow X_1 X_2 \dots X_n$. Здесь X_i — может быть как терминалом, так и нетерминалом.

Синтаксический анализ

$A \rightarrow XYZ$

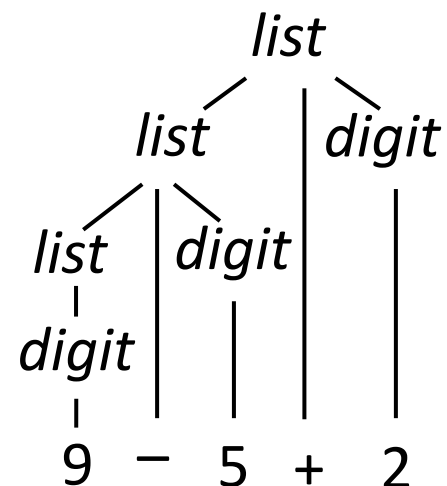


$list^* \rightarrow list + digit$

$list \rightarrow list - digit$

$list \rightarrow digit$

$digit \rightarrow 0|1|2|3|4|5|6|7|8|9$



Процесс поиска дерева разбора для данной строки терминалов называется *разбором* (parsing) или *синтаксическим анализом* этой строки.

Неоднозначности

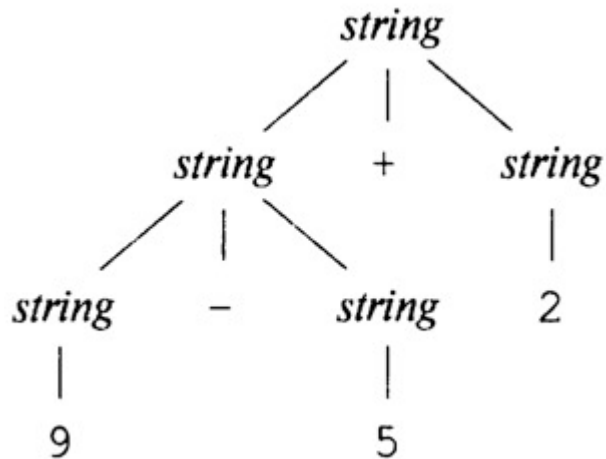
Исходная грамматика:

1: $list^* \rightarrow list + digit \mid list \rightarrow list - digit \mid list \rightarrow digit$

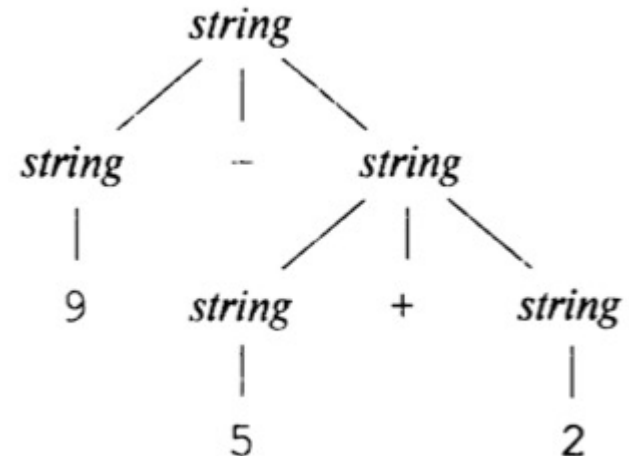
2: $digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Преобразованная (но не эквивалентная с точки зрения разбора):

$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



(9-5)+2



9-(5+2)

Ассоциативность операторов

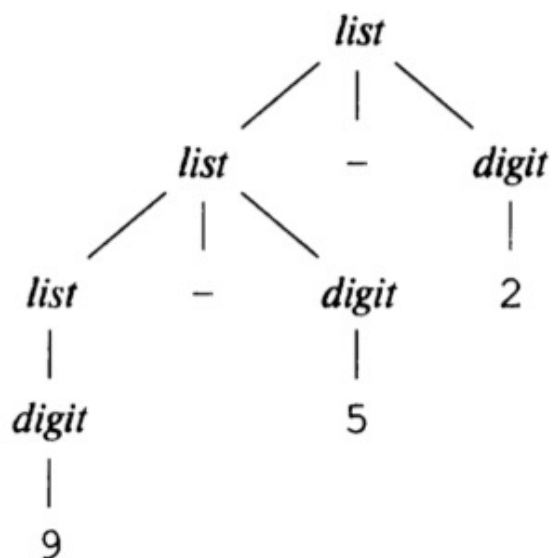
Левассоциативные

Примеры: +, -, *, /

$$a + b + c \Leftrightarrow (a + b) + c$$

$list \rightarrow list + digit$

$digit \rightarrow 0|1|2|3|4|5|6|7|8|9$



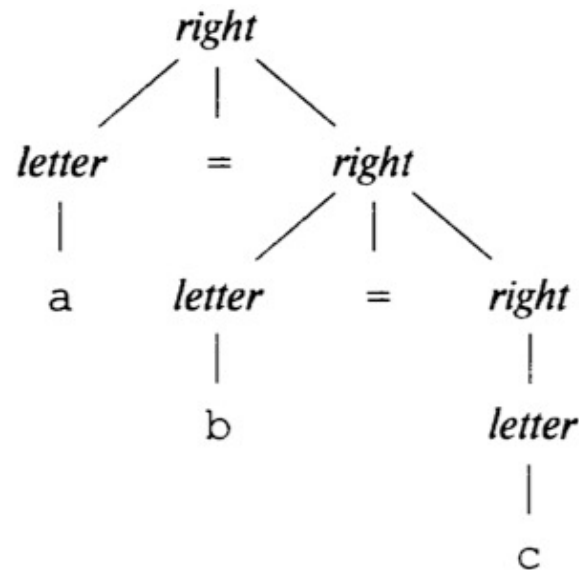
Правоассоциативные

Примеры: =

$$a = b = c \Leftrightarrow a = (b = c)$$

$right \rightarrow letter = right \mid letter$

$letter \rightarrow a \mid b \mid \dots \mid z$



Приоритет операторов

$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$

$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$

$\text{factor} \rightarrow \text{digit} \mid (\text{expr})$

Грамматика для подмножества Java

stmt $\rightarrow$ id = expression ;
| if (expression) stmt
| if (expression) stmt else stmt
| while (expression) stmt
| do stmt while (expression) ;
| { stmts }
stmts $\rightarrow$ stmts stmt
| ϵ

Синтаксически Управляемая Трансляция

При выполнении синтаксического анализа («парсинга») строки, можно связать с продукциями семантические правила (или программный код), и выполнять их при «свертке» соответствующих нетерминалов. Это и есть синтаксически управляемая трансляция.

Постфиксная запись

Пример:

$$(9 - 5) + 2 \Rightarrow 9\ 5 - 2 +$$

$$9 - (5 + 2) \Rightarrow 9\ 5\ 2 + -$$

Вычисление постфиксных выражений:

9 5 2 + - 3 *		9 5 2
9 5 2 + - 3 *		9 7
9 5 2 + - 3 *		2
9 5 2 + - 3 *		2 3
9 5 2 + - 3 *		6

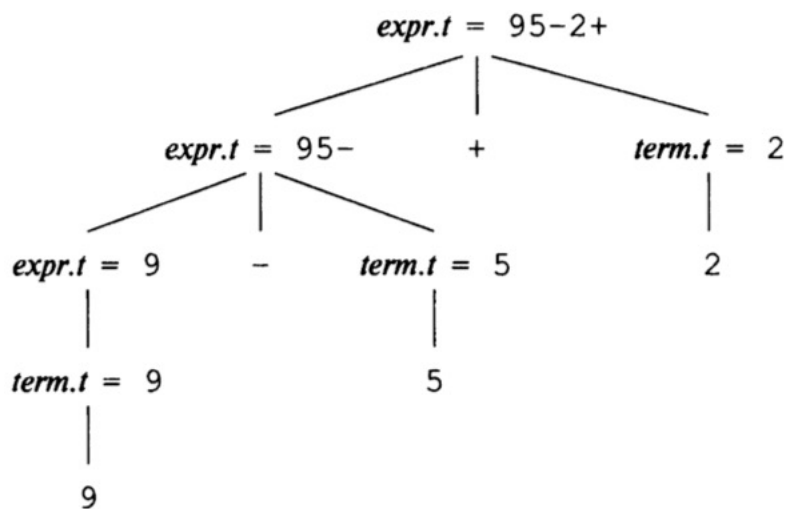
Строка сканируется слева направо, числа помещаются в стек. Когда встречаем оператор, аргументы берутся с вершины стека, операция выполняется, и значение помещается обратно на стек.

Постфиксная запись

Постфиксной записью для выражения E является:

1. Если E является переменной или константой, то постфиксная запись E представляет собой само E .
2. Если E — выражение вида $E_1 \text{ op } E_2$, где **op** — произвольный бинарный оператор, то постфиксная запись E представляет собой $E'_1 E'_2 \text{ op}$, 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 .

Синтезированные и унаследованные атрибуты



Синтезированный атрибут для символа X (вершины дерева разбора) вычисляется на основе атрибутов потомков и самого узла X

Унаследованные атрибуты вычисляются на основе атрибутов предков узла.

Синтаксически управляемые определения

Продукция

$expr \rightarrow expr_1 + term$

$expr \rightarrow expr_1 - term$

$expr \rightarrow term$

$term \rightarrow 0$

Семантическое правило

$expr.t = expr_1.t || term.t || '+'$

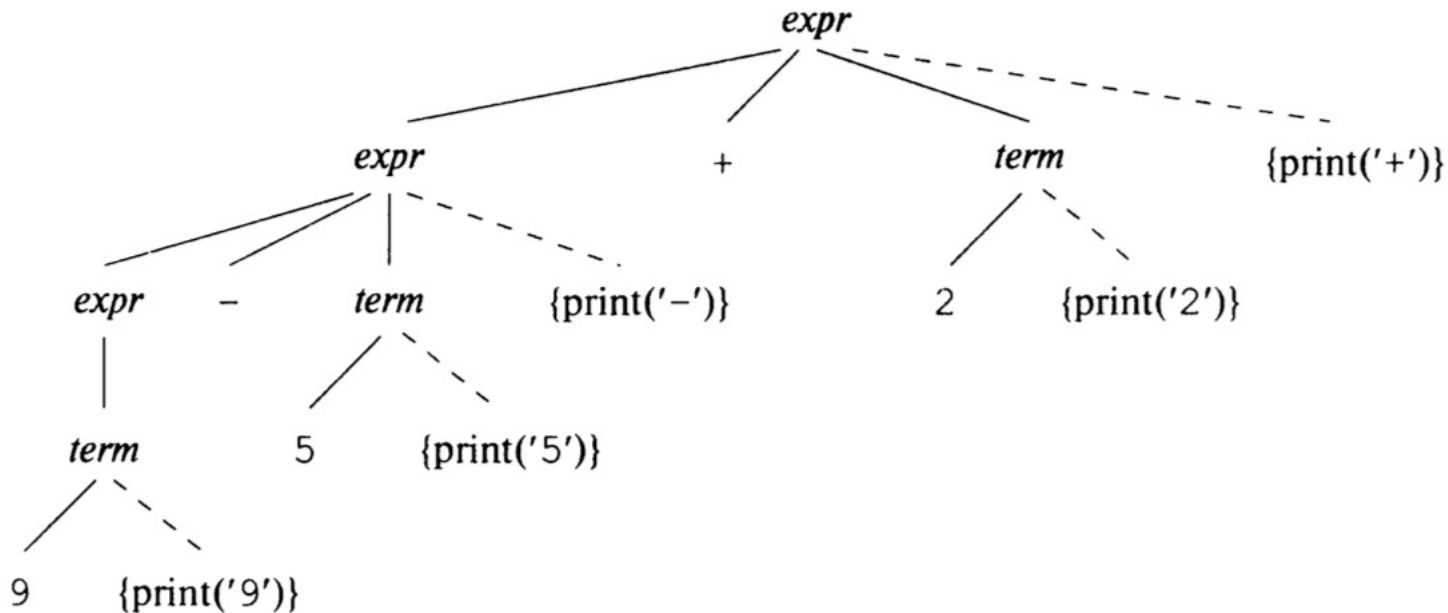
$expr.t = expr_1.t || term.t || '-'$

$expr.t = term.t$

$term.t = '0'$

|| – конкатенация строк

Схема Трансляции



Правила, дополненные семантическими действиями

$expr \rightarrow expr_1 + term \quad \{\text{print('+')}\}$

$expr \rightarrow expr_1 - term \quad \{\text{print('-')}\}$

$expr \rightarrow term$

$term \rightarrow 0 \quad \{\text{print('0')}\}$

Метод Рекурсивного Спуска

Пример грамматики

$stmt \rightarrow$ **expr ;**
 | **if (expr) stmt**
 | **for (optexpr ; optexpr ; optexpr) stmt**
 | **other**

$optexpr \rightarrow$ ϵ
 | **expr**

Метод Рекурсивного Спуска

```
stmt → for ( optexpr ; optexpr ; optexpr ) stmt
```

Предиктивный анализатор (predictive parsing): сканируемый символ однозначно определяет поток управления в теле процедуры для каждого нетерминала.

Последовательность вызовов процедур при обработке входной строки неявно определяет его дерево разбора.

```
match(for); match('(');  
optexpr(); match(';'); optexpr(); match(';'); optexpr();  
match(')');
```

Функция `match(t)`:

- сравнивает свой аргумент `t` со сканируемым символом и переходит к следующему символу в случае соответствия
- изменяет значение глобальной переменной `lookahead`, которая хранит сканируемый входной терминал

Метод Рекурсивного Спуска

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

$FIRST(\alpha)$ – множество терминалов, которые могут появиться в качестве первого символа одной или нескольких строк, сгенерированных из α . Если α может порождать ϵ , то ϵ также входит в $FIRST(\alpha)$

$FIRST(stmt) = \{\mathbf{expr}, \mathbf{if}, \mathbf{for}, \mathbf{other}\}$

Метод Рекурсивного Спуска

```
stmt  → expr ;  
      | if ( expr ) stmt  
      | for ( optexpr ; optexpr ; optexpr ) stmt  
      | other
```

```
optexpr →  $\epsilon$   
        | expr
```

```
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();  
        case other:  
            match(other); break;  
        default:  
            report("syntax error");  
    }  
}
```

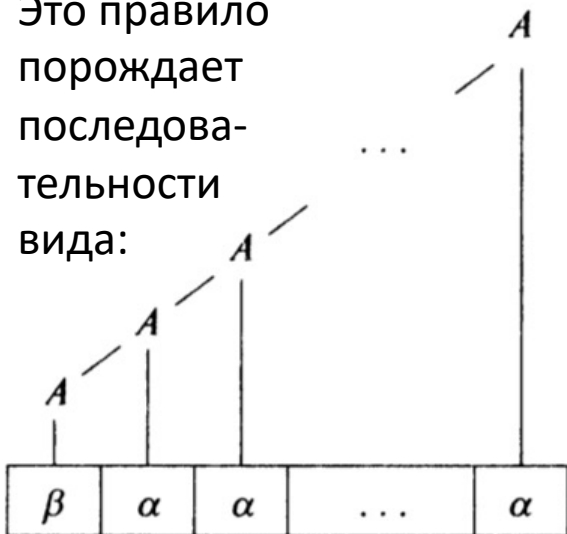
```
void optexpr() {  
    if (lookahead == expr)  
        match(expr);  
}  
  
void match(terminal t) {  
    if (lookahead == t)  
        lookahead = nextTerminal;  
    else report("syntax error");  
}
```

Устранениелевой Рекурсии

$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{term}$

Это леворекурсивное правило вида $A \rightarrow A \alpha \mid \beta$

Это правило порождает последовательности вида:



$A \rightarrow A \alpha \mid \beta$

$\Rightarrow$

Ту же последовательность можно получить и другим способом:



$\Rightarrow$

$A \rightarrow \beta R$
 $R \rightarrow \alpha R \mid \epsilon$

Устранение Лево́й Рекурсии в Схеме Трансляции

Удаление рекурсии: $A \rightarrow A \alpha \mid A \beta \mid \gamma \Rightarrow \begin{matrix} A \rightarrow \gamma R \\ R \rightarrow \alpha R \mid \beta R \mid \epsilon \end{matrix}$

Преобразуем схему трансляции:

$expr \rightarrow expr_1 + term \quad \{\text{print('+')}\}$
 $expr \rightarrow expr_1 - term \quad \{\text{print('-')}\}$
 $expr \rightarrow term$
 $term \rightarrow 0 \quad \{\text{print('0')}\}$

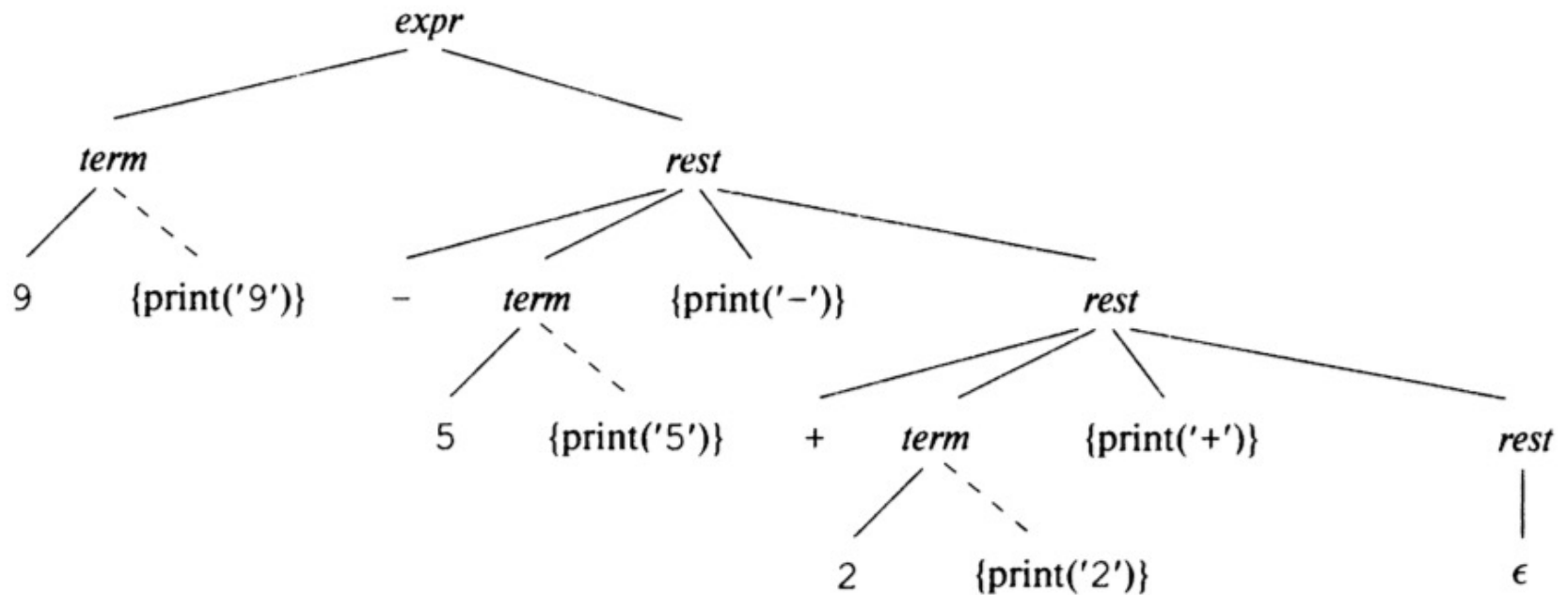
Учитывая, что

$A = expr$
 $\alpha = + term \{ \text{print('+')} \}$
 $\beta = - term \{ \text{print('-')} \}$
 $\gamma = term$



$expr \rightarrow term \ rest$
 $rest \rightarrow \begin{matrix} + term \{ \text{print('+')} \} rest \\ - term \{ \text{print('-')} \} rest \\ \epsilon \end{matrix}$
 $term \rightarrow 0 \{ \text{print('0')} \}$

Устранение Лево́й Рекурсии в Схеме Трансляции



$(9 - 5) + 2 \Rightarrow 9\ 5 - 2 +$

Реализация Схемы Синтаксически Управляемой Трансляции

```
expr  →  term rest  
  
rest  →  + term { print('+') } rest  
        |  - term { print('-') } rest  
        |  ε  
  
term  →  0 { print('0') }
```

```
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");  
}
```

Лексический Анализ

Лексический Анализ

Основная задача лексического анализа – разбиение входного потока символов на лексемы (токены).

Что делается в лексическом анализаторе:

- Удаление пробельных символов и комментариев
- Считывание констант
- Распознавание ключевых слов и идентификаторов
- Распознавание операторов
 - Опережающее чтение: >, >=, >>

Лексический Анализ

Почему лексический анализ выделяется в отдельную фазу?

- упрощение разработки
- увеличение эффективности компилятора

Примеры токенов:

ТОКЕН	НЕФОРМАЛЬНОЕ ОПИСАНИЕ	ПРИМЕРЫ ЛЕКСЕМ
if	Символы i, f	if
else	Символы e, l, s, e	else
comparison	< или > или <= или >= или == или !=	<=, !=
id	Буква, за которой следуют буквы и цифры	pi, score, D2
number	Любая числовая константа	3.14159, 0, 6.02e23
literal	Все, кроме ", заключенное в двойные кавычки	"core dumped"

Регулярные выражения

Цифра: **[0-9]**

Идентификатор: **[a-z][a-z0-9]***

Целое: **{DIGIT}+**

Вещественное: **{DIGIT}+"."{DIGIT}***

Ключевые слова: **if|then|begin|end|function**

Операторы: **+|-|*|/**

Комментарий (Паскаль): **{[^\\n]*}**

Пробельные символы: **[\\t\\n]+**

Взаимодействие Лексического Анализатора с Синтаксическим



Абстрактное Синтаксическое Дерево

Абстрактное Синтаксическое Дерево, Abstract Syntax Tree (**AST**).

В AST похожие операторы объединены в группы, т.к. во время генерации кода (или анализа) обрабатываются схожим образом.

КОНКРЕТНЫЙ СИНТАКСИС	АБСТРАКТНЫЙ СИНТАКСИС
=	assign
	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	op
* / %	op
!	not
^{-unary}	minus
[]	access

Построение Абстрактного Синтаксического Дерева (AST)

Пусть у каждого символа грамматики задан атрибут **n**, в котором мы будем хранить ссылку на соответствующий ему узел в AST. Пусть **X.n** – указатель на базовый тип *ASTNode* *, а каждому оператору соответствует унаследованный от него класс с собственным конструктором. Мы будем конструировать AST во время синтаксически управляемой трансляции с использованием подобной схемы:

$add \rightarrow add_1 + term$	$\{ add.n = \mathbf{new} Op('+', add_1.n, term.n); \}$
$\quad \quad term$	$\{ add.n = term.n; \}$
$term \rightarrow term_1 * factor$	$\{ term.n = \mathbf{new} Op('*', term_1.n, factor.n); \}$
$\quad \quad factor$	$\{ term.n = factor.n; \}$
$factor \rightarrow (expr)$	$\{ factor.n = expr.n; \}$
$\quad \quad \mathbf{num}$	$\{ factor.n = \mathbf{new} Num(\mathbf{num.value}); \}$

Построение Абстрактного Синтаксического Дерева (AST)

$program \rightarrow block \quad \{ \text{return } block.n; \}$

$block \rightarrow \{ ' \{ ' stmts ' \} ' \quad \{ block.n = stmts.n; \}$

$stmts \rightarrow stmts_1 stmt \quad \{ stmts.n = \text{new Seq}(stmts_1.n, stmt.n); \}$
 $\quad \mid \epsilon \quad \{ stmts.n = \text{null}; \}$

$stmt \rightarrow expr ; \quad \{ stmt.n = \text{new Eval}(expr.n); \}$
 $\quad \mid \text{if } (expr) stmt_1 \quad \{ stmt.n = \text{new If}(expr.n, stmt_1.n); \}$
 $\quad \mid \text{while } (expr) stmt_1 \quad \{ stmt.n = \text{new While}(expr.n, stmt_1.n); \}$
 $\quad \mid \text{do } stmt_1 \text{ while } (expr) ; \quad \{ stmt.n = \text{new Do}(stmt_1.n, expr.n); \}$
 $\quad \mid block \quad \{ stmt.n = block.n; \}$

Статические проверки

Выполняются на этапе построения дерева (либо во время генерации трехадресного представления):

- **Проверки синтаксиса.** Правила, которые нельзя отразить в грамматике – например, что идентификатор должен быть объявлен в области видимости не более одного раза, или что инструкция `break` располагается в охватывающем цикле или инструкции `switch`.
- **Проверки типов.** Оператор применен к корректному количеству операторов допустимого типа. Также программа проверки типов может вставить оператор преобразования типов в синтаксическое дерево (например, при сложении целого числа с числом с плавающей точкой).

Статические проверки

Примеры статических проверок:

- Слева от присваивания должно находиться lvalue (должно быть ячейкой памяти), например:

```
i = 5;  
i = i + 1;  
a[i] = *p;
```

- Проверка типов:
Правило грамматики:

```
if ( expr ) stmt
```

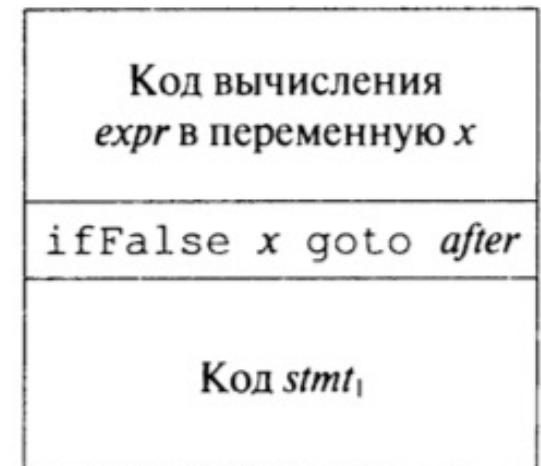
Проверка:

```
if (  $E_1.type == E_2.type$  )  $E.type = \text{boolean};$   
else error;
```

Трансляция Инструкций

Код для трансляции условного оператора: **if** (*expr*) **then** *stmt*₁

```
class If extends Stmt {  
    Expr E; Stmt S;  
    public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }  
    public void gen() {  
        Expr n = E.rvalue();  
        emit( "ifFalse " + n.toString() + " goto " + after);  
        S.gen();  
        emit(after + ":" );  
    }  
}
```



after →

Трансляция Выражений

Пример:

```
a[i] = 2*a[j-k]    ⇒    t3 = j - k
                        t2 = a [ t3 ]
                        t1 = 2 * t2
                        a [ i ] = t1
```

Трансляция Выражений

Трансляция левой части выражений:

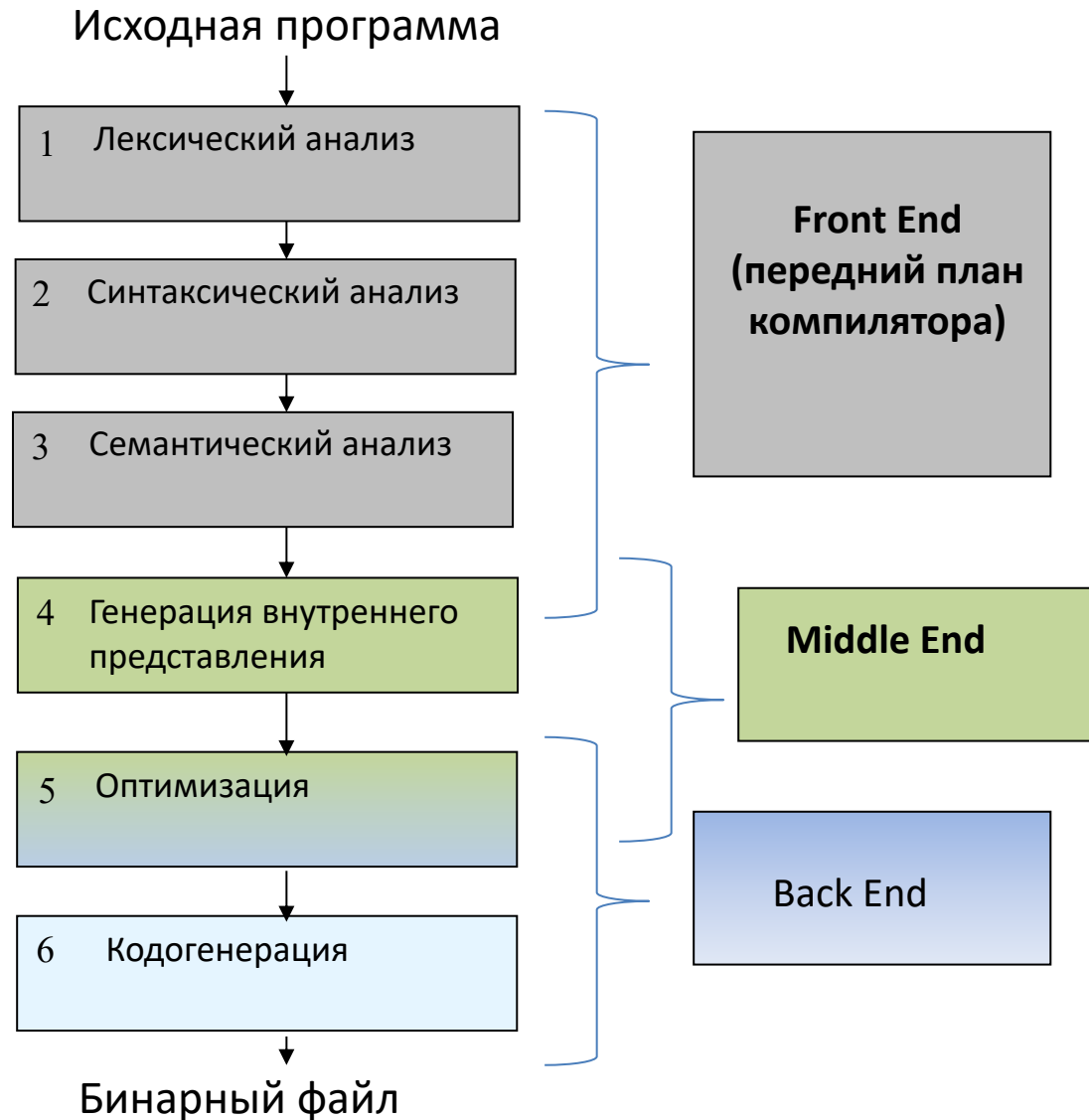
```
Expr lvalue(x : Expr) {  
    if ( x является узлом Id ) return x;  
    else if ( x является узлом Access (y, z), а y — узел Id ) {  
        return new Access (y, rvalue(z));  
    }  
    else error;  
}
```


Трансляция Выражений

Трансляция правой части выражений:

```
Expr rvalue(x : Expr) {  
  if ( x — узел Id или Constant ) return x;  
  else if ( x — узел Op (op, y, z) или Rel (op, y, z) ) {  
    t = новая временная переменная;  
    Генерация строки для t = rvalue(y) op rvalue(z);  
    return Новый узел t;  
  }  
  else if ( x узел Access (y, z) ) {  
    t = новая временная переменная;  
    Вызов lvalue(x) возвращающий Access (y, z');  
    Генерация строки для t = Access (y, z');  
    return Новый узел t;  
  }  
  else if ( x — узел Assign (y, z) ) {  
    z' = rvalue(z);  
    Генерация строки для lvalue(y) = z';  
    return z';  
  }  
}
```

Фазы компиляции



Bison

Грамматика для вычисления выражений в постфиксной записи (`rpcalc.y`):

```
input:      /* empty */
          | input line
;

line:       '\n'
          | exp '\n' { printf ("\t%.10g\n", $1); }
;

exp:        NUM          { $$ = $1; }
          | exp exp '+'   { $$ = $1 + $2; }
          | exp exp '-'   { $$ = $1 - $2; }
          | exp exp '*'   { $$ = $1 * $2; }
          | exp exp '/'   { $$ = $1 / $2; }
          /* Exponentiation */
          | exp exp '^'   { $$ = pow ($1, $2); }
          /* Unary minus */
          | exp 'n'       { $$ = -$1; }
;

%%
```

Bison

Определения типов и токены :

```
%{  
#define YYSTYPE double  
#include <math.h>  
%}
```

```
%token NUM
```

```
%% /* Grammar rules and actions follow */
```

Bison

Лексический анализатор:

```
#include <ctype.h>

yylex ()
{
    int c;

    /* skip white space */
    while ((c = getchar ()) == ' ' || c == '\t')
        ;
    /* process numbers */
    if (c == '.' || isdigit (c))
    {
        ungetc (c, stdin);
        scanf ("%lf", &yylval);
        return NUM;
    }
    /* return end-of-file */
    if (c == EOF)
        return 0;
    /* return single chars */
    return c;
}
```

Bison

Управляющий файл:

```
main ()  
{  
    yyparse ();  
}
```

Flex

```
%{
/* need this for the call to atof() below */
#include <math.h>
}%

DIGIT      [0-9]
ID          [a-z][a-z0-9]*

%%

{DIGIT}+    {
    printf( "An integer: %s (%d)\n", yytext,
            atoi( yytext ) );
}

{DIGIT}+"."{DIGIT}*    {
    printf( "A float: %s (%g)\n", yytext,
            atof( yytext ) );
}

if|then|begin|end|procedure|function    {
    printf( "A keyword: %s\n", yytext );
}

{ID}        printf( "An identifier: %s\n", yytext );
```

Flex

```
"+"|"-"|"*"|"/"    printf( "An operator: %s\n", yytext );

"{ "[^]\n]*" }      /* eat up one-line comments */

[ \t\n]+             /* eat up whitespace */

.                    printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
{
    ++argv, --argc; /* skip over program name */
    if ( argc > 0 )
        yyin = fopen( argv[0], "r" );
    else
        yyin = stdin;

    yylex();
}
```