# 1-ый вариант

1. Дайте определение понятиям «компилятор» и «интерпретатор». Перечислите их основные этапы работы и составные части, выполняющие эти этапы.

**Определение понятий**

1. **Компилятор** — это программа, которая переводит код, написанный на человекочитаемом языке программирования, в машинный код, который может быть непосредственно выполнен процессором компьютера. Компилятор работает с исходным кодом, анализирует его структуру и синтаксис, выполняет оптимизации, а затем создает исполняемый файл.
2. **Интерпретатор** — это программа, которая непосредственно выполняет код на языке программирования, интерпретируя его строки по мере выполнения. При этом исходный код не преобразуется в отдельный исполняемый файл; инструкции обрабатываются последовательно.

**Основные этапы работы и составные части**

**Компилятор**

Компилятор включает следующие этапы работы, которые делятся на **анализ** (front-end) и **синтез** (back-end):

1. **Лексический анализ**:
   - Деление исходного кода на "слова" или токены, например идентификаторы, ключевые слова, знаки операций.
   - Выполняется лексическим анализатором.
2. **Синтаксический анализ**:
   - Построение синтаксического дерева, которое отражает структуру программы.
   - Выполняется синтаксическим анализатором.
3. **Семантический анализ**:
   - Проверка допустимости операций, корректности типов и других логических правил языка.
   - Выполняется семантическим анализатором.
4. **Генерация промежуточного представления**:
   - Создание промежуточного низкоуровневого кода, который удобен для дальнейших оптимизаций.
   - Генератор промежуточного кода выполняет эту задачу.
5. **Оптимизация**:
   - Улучшение производительности кода, например путем устранения избыточных операций.
   - Оптимизатор выполняет эту задачу.
6. **Генерация машинного кода**:
   - Преобразование промежуточного представления в машинные инструкции, понятные процессору.
   - Генератор машинного кода выполняет эту задачу.
7. **Компоновка**:

- Объединение модулей программы и библиотек в единый исполняемый файл.
- Выполняется компоновщиком.

**Интерпретатор**

Интерпретатор выполняет следующие этапы:

1. **Чтение исходного кода**:
   - Исходный код поступает на вход интерпретатору.
2. **Лексический анализ**:
   - Аналогично компилятору, программа делится на токены.
   - Выполняется лексическим анализатором.
3. **Синтаксический анализ**:
   - Построение структуры программы.
   - Выполняется синтаксическим анализатором.
4. **Выполнение кода**:
   - Код выполняется по мере анализа. Каждый оператор преобразуется в действия, которые сразу исполняются.
   - Выполняется интерпретирующей машиной.

---

Источник: информация взята из лекции 11, файл «01-11-2024-ib_lect11.pdf»

## 2. Что такое алгебраический тип данных? Дайте определение. Какие средства (процедуры, макросы, структуры данных) надо ввести, чтобы иметь возможность определять собственные алгебраические типы в программе на языке с динамической типизацией? Какие процедуры должны быть определены для каждого алгебраического типа?

**Определение алгебраического типа данных**

**Алгебраический тип данных** — это составной тип данных, который определяется как объединение (сумма) или пересечение (произведение) других типов данных. Такие типы данных используются для представления структурированной информации, где значения могут принадлежать различным подтипам (в случае суммы) или представлять комбинацию нескольких полей (в случае произведения).

Примеры:

1. Сумма типов: значение может быть либо числом, либо строкой, либо списком.
2. Произведение типов: структура, содержащая поля с определенными

типами (например, точка с координатами x и y).

**Средства для реализации алгебраических типов данных в языке с динамической типизацией**

Для реализации алгебраических типов данных в языке с динамической типизацией, например, в Scheme, необходимы следующие элементы:

1. **Конструкторы**:
   - Процедуры для создания значений определенных типов.
   - Например, для структуры точки можно использовать (make-point x y).
2. **Селекторы**:
   - Процедуры для доступа к отдельным полям структуры.
   - Например, (point-x point) для доступа к полю x структуры точки.
3. **Типовые предикаты**:
   - Процедуры, проверяющие принадлежность значения определенному типу.
   - Например, (point? value) проверяет, является ли value точкой.
4. **Сопоставление с образцом (pattern matching)**:
   - Используется для обработки значений, принадлежащих различным подтипам алгебраического типа.
   - Может быть реализовано через макросы cond, case или пользовательские конструкции.

**Реализация алгебраических типов в программе**

В Scheme можно использовать макросы, списки, ассоциативные списки или замыкания для представления алгебраических типов данных:

**Пример: структура "Точка"**

```
;; Конструктор
(define (make-point x y)
  (list 'point x y))

;; Селекторы
(define (point-x point)
  (cadr point))

(define (point-y point)
  (caddr point))

;; Предикат типа
(define (point? value)
  (and (list? value) (eq? (car value)
'point)))
```

**Пример: сумма типов (выражения)**

```
;; Конструкторы
(define (make-number n)
  (list 'number n))

(define (make-add expr1 expr2)
  (list 'add expr1 expr2))

;; Предикат типа
(define (number? expr)
  (and (list? expr) (eq? (car expr)
'number)))

(define (add? expr)
  (and (list? expr) (eq? (car expr)
'add)))

;; Сопоставление с образцом
(define (evaluate expr)
```

```
  (cond
    ((number? expr) (cadr expr))
    ((add? expr) (+ (evaluate (cadr
expr))
                    (evaluate (caddr expr))))
    (else (error "Unknown expression
type"))))
```

### Процедуры, которые должны быть определены для каждого алгебраического типа

1. **Конструкторы**:
    - Процедуры создания значений (для всех подтипов).
2. **Селекторы**:
    - Процедуры доступа к полям структур (для типов-произведений).
3. **Предикаты типов**:
    - Проверка принадлежности значений конкретному подтипу.
4. **Процедуры обработки**:
    - Для работы с суммой типов: функции или макросы для анализа типа и обработки значений (например, сопоставление с образцом).
    - Для работы с произведением типов: процедуры для выполнения операций над компонентами.

---

Источник: информация взята из лекций 7 и 3, файлы «05-10-2024-ib_lect07.pdf» и «20-09-2024-ib_lect03.pdf»

## 3. Используя БНФ, запишите грамматику для телефонных номеров, записанных в формате +код-страны(код-города)код-станции–номер. Подчеркните терминальные символы. Приведите пример последовательности символов, соответствующей этой грамматике. Покажите лексемы, на которые согласно этой грамматике будет разбита исходная последовательность символов.

#### Грамматика телефонных номеров в БНФ

В формате **+код-страны(код-города)код-станции–номер** мы можем выделить следующие составляющие:

1. Код страны: последовательность цифр после +.

2. Код города: последовательность цифр в круглых скобках.
3. Код станции: последовательность цифр после дефиса.
4. Номер: последовательность цифр после длинного тире.

**Грамматика в БНФ:**

<TelephoneNumber> ::= "+" <CountryCode> "(" <CityCode> ")" <StationCode> "–" <Number>
<CountryCode> ::= <Digit>+
<CityCode> ::= <Digit>+
<StationCode> ::= <Digit>+
<Number> ::= <Digit>+
<Digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

**Терминальные символы:**

- +, (, ), –
- Символы 0-9.

**Пример телефонного номера**

**Пример:** +7(495)123-4567

**Лексемы:**

1. \+ — знак, обозначающий международный формат.
2. 7 — код страны.
3. 495 — код города.
4. 123 — код станции.
5. 4567 — номер.

---

Основы описания грамматик (включая терминальные и нетерминальные символы, правила грамматики, использование БНФ) взяты из лекции 13, файл «08-11-2024-ib_lect13.pdf»

4. Используя БНФ, запишите грамматику логических выражений, записанных в традиционной инфиксной нотации с использованием бинарных операторов and, or и унарного оператора not. Операндами могут быть имена переменных и константы true и false. Грамматика должна учитывать приоритет операций. Порядок вычислений может быть также задан круглыми скобками. Приведите пример выражения и дерева его разбора.

**Грамматика логических выражений в БНФ**

Грамматика учитывает приоритет операций:

1. **not** — унарный оператор с наивысшим приоритетом.
2. **and** — бинарный оператор с приоритетом выше, чем or.
3. **or** — бинарный оператор с самым низким приоритетом.
4. Скобки () задают явный порядок вычислений.

**БНФ для логических выражений:**

<Expression> ::= <OrExpression>
<OrExpression> ::= <AndExpression> | <OrExpression> "or" <AndExpression>
<AndExpression> ::= <NotExpression> | <AndExpression> "and" <NotExpression>
<NotExpression> ::= "not" <Primary> | <Primary>
<Primary> ::= "true" | "false" | <Variable> | "(" <Expression> ")"
<Variable> ::= <Letter> | <Letter> <Variable>
<Letter> ::= "a" | "b" | "c" | ... | "z"

**Терминальные символы:**

- "or", "and", "not", "true", "false", (, ).
- Символы a-z.

---

**Пример логического выражения**

**Выражение:**
not (a and b) or true

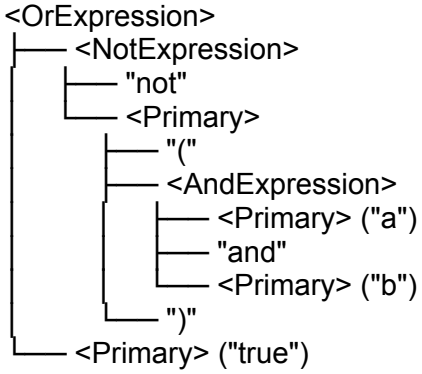
---

**Дерево разбора**

1. **Корень дерева** — <OrExpression>.
2. Дочерние узлы:
    - Левый узел — <NotExpression> (разбирает not (a and b)).
    - Правый узел — <Primary> (разбирает true).

---

Дерево:



**Разъяснение**

1. **not (a and b)**:
   - not применяется к <Primary>, заключенному в скобки.
   - Внутри скобок разбирается <AndExpression>: переменные a и b, соединенные оператором and.
2. **or true**:
   - Оператор or соединяет результат от not (a and b) с <Primary> (true).

---

Источник: информация взята из лекции 13, файл «08-11-2024-ib_lect13.pdf»

## 5. Какая из приведенных ниже грамматик является LL(k)-грамматикой? Можно ли непосредственно на основе этой грамматики реализовать нисходящий рекурсивный парсер? Почему? Приведите пример исходной последовательности и синтаксического дерева для этой грамматики.

а) A → BA | ε
B → x | y A z

б) A → x A | B
B → x B y | ε

Грамматика (а):

**Наличие неоднозначности:**

- Для *A*: Если *FIRST(B)* содержит *x*, *y*, и *ε*, то возможен конфликт при выборе *A→BA* или *A→ε*.
- Для *B*: Появляется зависимость от контекста *A*, так как для *B→yAz* требуется анализировать *A*, что делает *FIRST* неопределённым по *k=1*.

**Вывод:** Грамматика не является LL(k) (даже при *k>1*) из-за неоднозначного выбора между правилами.

**Возможность создания**

Грамматика (б):

- **Левая рекурсия:**
  - *A→xA*: Налицо левая рекурсия, которая делает грамматику не LL(1).
- **Возможность преобразования:**
  - Устранение левой рекурсии может сделать грамматику подходящей для LL(1), но в текущем виде она не является LL(1).

**Возможность создания рекурсивного нисходящего**

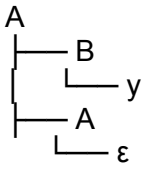
| **рекурсивного нисходящего парсера:** | **парсера:** |
| --- | --- |
| Непосредственно создать парсер невозможно из-за неоднозначности выбора правил. | Требуется устранение левой рекурсии, иначе парсер не сможет работать корректно. |

**Пример исходной последовательности и синтаксического дерева**

**Для грамматики (а):**

Исходная последовательность: y x z.

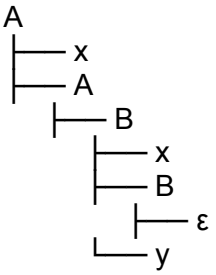
Синтаксическое дерево:




**Для грамматики (б):**

Исходная последовательность: x x y.

Синтаксическое дерево:




**Итог**

- **Грамматика (а):** Не LL(k), невозможно реализовать парсер без преобразования.
- **Грамматика (б):** Не LL(k) из-за левой рекурсии, требуется устранение левой рекурсии перед созданием парсера.

---

Источник: Ответ основан на лекции 13, файл «08-11-2024-ib_lect13.pdf»

# 2-ый вариант

## 1. Дайте определение понятию «грамматика». В его контексте определите понятия «терминальный символ», «нетерминальный символ», «правило грамматики».

**Определение грамматики**

**Грамматика** — это формальный способ описания синтаксиса формального языка. Она определяет правила построения строк, принадлежащих языку, с использованием набора символов.

Грамматика состоит из следующих компонентов:

1. **Аксиомы** — начальный символ, с которого начинается процесс порождения строк.
2. **Множества терминальных символов** — конечные символы, из которых состоят строки языка.
3. **Множества нетерминальных символов** — промежуточные символы, которые раскрываются в процессе применения правил грамматики.
4. **Множества правил грамматики** — инструкции, определяющие, как раскрываются нетерминальные символы.

---

**Терминальные и нетерминальные символы**

1. **Терминальный символ**:
   - Это символ алфавита языка, который является конечным и не подлежит дальнейшему раскрытию.
   - Пример: в языке программирования терминальными символами могут быть if, +, x, 1.
2. **Нетерминальный символ**:
   - Это символ, который раскрывается согласно правилам грамматики и преобразуется в последовательность других символов (терминальных и/или нетерминальных).
   - Пример: <Expression>, <Statement>.

---

**Правило грамматики**

**Правило грамматики** — это инструкция вида $A \rightarrow \alpha$, где:

- $A$ — нетерминальный символ.
- $\alpha$ — последовательность терминальных и/или нетерминальных символов.

Правило описывает, как заменить $A$ на $\alpha$ в процессе порождения строк.

---

**Пример грамматики**

Для описания арифметических выражений можно задать грамматику:

1. Терминальные символы: +, *, n (число), (, ).
2. Нетерминальные символы: <Expression>, <Term>, <Factor>.
3. Правила:

   <Expression> ::= <Term> | <Expression> + <Term>

   <Term> ::= <Factor> | <Term> * <Factor>

   <Factor> ::= n | ( <Expression> )

---

Источник: Информация взята из лекции 13, файл «08-11-2024-ib_lect13.pdf»

2. Используя БНФ, запишите грамматику для списка слов, разделенных запятыми. В любом месте последовательности, кроме как в середине слов, могут встречаться пробельные символы, которые должны игнорироваться. Слова могут состоять только из букв латинского алфавита. Подчеркните терминальные символы. Приведите пример последовательности символов, соответствующей этой грамматике. Покажите лексемы, на которые согласно этой грамматике будет разбита исходная последовательность символов.

**Грамматика для списка слов в БНФ**

Учитываем, что:

1. Список состоит из слов, разделенных запятыми.
2. Пробельные символы игнорируются, кроме случаев внутри слова.
3. Слова содержат только буквы латинского алфавита.

**Грамматика:**
<List> ::= <Word> | <Word> "," <List> | <Whitespace> <List> <Whitespace>
<Word> ::= <Letter> | <Letter> <Word>
<Letter> ::= "a" | "b" | "c" | ... | "z" | "A" | "B" | "C" | ... | "Z"
<Whitespace> ::= " " | <Whitespace> " "

**Терминальные символы:**

- Запятая ,.
- Пробельный символ " ".
- Буквы латинского алфавита a-z, A-Z.

---

**Пример последовательности символов**

**Пример:**
apple , banana,carrot , dog

---

**Лексемы**

Исходная последовательность разбивается на следующие лексемы:

1. apple — слово.
2. , — разделитель.
3. banana — слово.
4. , — разделитель.
5. carrot — слово.
6. , — разделитель.
7. dog — слово.

---

**Разъяснение**

1. **Пропуск пробелов:**
   Пробелы перед, после или между словами (вне слов) игнорируются согласно определению <Whitespace>.
2. **Слова и разделители:**
   Слова определяются как последовательность букв <Letter>, а запятые разделяют слова в списке.

---

**Подробная структура**

Исходная строка:
apple , banana,carrot , dog

После анализа:

```
<List>
├── <Word> ("apple")
├── ","
├── <List>
│   ├── <Word> ("banana")
│   ├── ","
│   ├── <List>
│   │   ├── <Word> ("carrot")
```

```
│       ├── ","
│       └── <List>
│           └── <Word> ("dog")
```

---

Данный ответ основан на анализе и правилах формальных грамматик, описанных в лекции 13, файл «08-11-2024-ib_lect13.pdf»

3. Используя БНФ, запишите грамматику выражений с операциями над списками. Выражения записаны в традиционной инфиксной нотации с использованием бинарных операторов конкатенации двух списков (+) и присоединения элемента в начало списка (:). Операндами могут быть имена переменных и константа () (пустой список). Приоритет операторов одинаков. Операторы являются правоассоциативными. Порядок вычислений может быть задан круглыми скобками. Приведите пример выражения и дерева его разбора.

**Грамматика выражений с операциями над списками в БНФ**

Учитываем, что:

1. Выражения записаны в инфиксной нотации.
2. Операции + (конкатенация списков) и : (добавление элемента в начало списка) имеют одинаковый приоритет.
3. Операторы являются **правоассоциативными**.
4. Операндами могут быть имена переменных и пустой список ().

**Грамматика:**

```
<Expression> ::= <Term> | <Term> <Operator> <Expression>
<Term> ::= <Variable> | "()" | "(" <Expression> ")"
<Operator> ::= "+" | ":"
<Variable> ::= <Letter> | <Letter> <Variable>
<Letter> ::= "a" | "b" | "c" | ... | "z"
```

**Терминальные символы:**

- Символы операторов: +, :.
- Пустой список: ().
- Круглые скобки: (, ).
- Буквы латинского алфавита a-z.

---

**Пример выражения**

**Выражение:**

a : b + c : ()

---

**Дерево разбора**

1. Корень дерева соответствует <Expression>.
2. Разбор начинается с самого правого оператора, так как операция является **правоассоциативной**.

**Дерево:**

<Expression>
├── <Term> ("a")
├── ":"
└── <Expression>
    ├── <Term> ("b")
    ├── "+"
    └── <Expression>
        ├── <Term> ("c")
        ├── ":"
        └── <Term> ("()")

---

**Объяснение разбора**

1. Выражение разбирается справа налево:
   - Самый правый оператор : разбирает c : ().
   - Следующий оператор + соединяет результат b с выражением c : ().
   - Самый левый оператор : разбирает a : ....
2. Лексемы, на которые разбивается выражение:
   - a, :, b, +, c, :, ().

Эта грамматика и дерево разбора обеспечивают правильное выполнение правоассоциативных операций с одинаковым приоритетом.

---

Ответ основан на принципах, описанных в лекции 13, файл «08-11-2024-ib_lect13.pdf». В лекции рассматриваются формальные грамматики, синтаксические деревья, правила ассоциативности и приоритета операций

4. Какая из приведенных ниже грамматик является LL(1)-грамматикой? Можно ли непосредственно на основе этой грамматики реализовать нисходящий рекурсивный парсер? Почему? Приведите пример исходной последовательности и синтаксического дерева для этой грамматики.

а) A → x B y

б) A → B | x y

B → A C
C → z B | A

B → x t y

**Грамматика (а):**

Анализ:

1. Проблемы с рекурсией:
   - B → A C → при подстановке A → x B y возникает цепочка:

     B → x B y C

   - Это указывает на левую рекурсию через B, что делает грамматику не LL(1).
   - Аналогичная проблема возникает в C → A.
2. Неоднозначность FIRST:
   - Для C → z B | A, множество FIRST(C) зависит от A (и, следовательно, от B), что делает анализ невозможным по одному символу.

Вывод:

- Грамматика не является LL(1) из-за левой рекурсии и неоднозначности выбора правила.

**Грамматика (б):**

Анализ:

1. Отсутствие левой рекурсии:
   - Нет прямой или косвенной левой рекурсии.
2. Определенность FIRST:
   - Для A → B | x y, FIRST(B) и FIRST(x y) определяются уникально:
     - FIRST(B) начинается с x (от правила B → x t y).
     - FIRST(x y) начинается с x (от x).
     - При наложении множеств для обоих правил возникает пересечение.
3. Неоднозначность FOLLOW:
   - Если B заканчивается на y, то символы после A могут не позволить однозначно выбрать, какой путь был использован: A → B или A → x y.

Вывод:

- Грамматика не является LL(1) из-за пересечения в FIRST для альтернатив A → B | x y.

---

Возможность создания нисходящего рекурсивного парсера

Нельзя создать парсер напрямую, так как присутствует левая рекурсия.

Прямая реализация парсера невозможна из-за неоднозначности FIRST для правил.

---

Пример и дерево разбора

Исходная последовательность: x z x y y

- Разбор невозможен из-за левой рекурсии.

Исходная последовательность: x t y

- Невозможно однозначно разобрать, использовалось ли правило A → B или A → x y.

---

**Итог**

Ни одна из грамматик **не является LL(1)**. Обе требуют преобразования (например, устранения левой рекурсии и неоднозначности) перед созданием нисходящего рекурсивного парсера.

---

Источник: анализ основан на материалах лекции 13, файл «08-11-2024-ib_lect13.pdf»

## 5. Какой вычислительный процесс — линейно- или древовидно-рекурсивный имеет место в случае: (а) синтаксического анализа, (б) лексического анализа? Почему?

**Синтаксический анализ**

**Процесс:**

Синтаксический анализ преобразует последовательность токенов (результат лексического анализа) в **синтаксическое дерево** или **дерево разбора**. Каждый уровень грамматики, включая вложенные структуры, разбирается рекурсивно.

**Характер процесса:**

**Древовидно-рекурсивный**.

**Почему?**

1. **Построение дерева**:
    - Каждое правило

**Лексический анализ**

**Процесс:**

Лексический анализ разделяет исходную строку на последовательность токенов, используя **правила разметки**, заданные с помощью регулярных выражений.

**Характер процесса:**

**Линейный**.

**Почему?**

1. **Обработка последовательности символов**:

грамматики раскрывается в подузлы, и процесс продолжается рекурсивно для каждого нетерминального символа.
    - Например, если грамматика определяет выражение как <Expression> ::= <Term> "+" <Expression>, разбор <Expression> приводит к созданию подузлов для <Term> и нового <Expression>.
2. **Рекурсивная структура грамматики**:
    - Большинство контекстно-свободных грамматик подразумевают вложенные конструкции (например, скобки), что требует древовидного подхода.

- Лексический анализ обрабатывает входной текст символ за символом, сопоставляя их с паттернами регулярных выражений для токенов.
- Он не зависит от вложенных структур, а только от определения токена.

2. **Отсутствие вложенности**:
    - Лексический анализ не требует построения дерева или возврата к уже разобранным токенам, так как он работает только с последовательной строкой символов.

**Итог**

1. **Синтаксический анализ**: древовидно-рекурсивный процесс, поскольку требуется построение синтаксического дерева.
2. **Лексический анализ**: линейный процесс, так как вход обрабатывается последовательно без вложенных структур.

---

Источник: информация основана на материалах лекции 13, файл «08-11-2024-ib_lect13.pdf»