Translation methods

1. Грамматика: определение, типы грамматик по Хомскому, примеры.

Грамматика: определение, типы грамматик по Хомскому, примеры.

Грамматика - это система правил, которая определяет структуру и правильное использование языка. Грамматика обычно описывает синтаксические и семантические правила языка, определяет различные типы фраз и предложений, а также правила их образования.

По классификации Хомского существуют четыре основных типа грамматик:

1. Тип 0 (нерегулярная грамматика или неограниченная грамматика): Это самый общий тип грамматики, который не накладывает ограничений на правила грамматики. Он может описывать любой язык. Примером такой грамматики является машина Тьюринга, которая может моделировать произвольные вычисления.
2. Тип 1 (контекстно-зависимая грамматика): В этом типе грамматики левая часть правила может содержать один или несколько символов, а правая часть может содержать любую последовательность символов. Однако контекстно-зависимая грамматика имеет ограничение, что левая часть правила не может быть пустой. Примером такой грамматики является язык арифметических выражений.
3. Тип 2 (контекстно-свободная грамматика): В этом типе грамматики левая часть правила может содержать только один символ, а правая часть может содержать любую последовательность символов. Контекстно-свободные грамматики широко используются для описания синтаксиса языков программирования и компиляторов. Примером такой грамматики является грамматика для описания структуры простого арифметического выражения.
4. Тип 3 (регулярная грамматика): Этот тип грамматики является самым ограниченным. Он описывает самые простые языки, называемые регулярными языками. Регулярные грамматики часто используются для построения регулярных выражений и конечных автоматов. Примером регулярной грамматики является грамматика для описания языка, состоящего из последовательности букв.

Это основные типы грамматик по Хомскому с их примерами. Каждый тип грамматики имеет различные ограничения на структуру и правила языка, которые можно описать с помощью этой грамматики.

2. Дерево разбора. Однозначность грамматики. Примеры однозначных и неоднозначных грамматик.

Дерево разбора - это структура данных, которая представляет синтаксический анализ предложения или фразы в соответствии с правилами грамматики. Оно отображает разбор каждого элемента предложения и связи между ними.

Однозначность грамматики означает, что для любого предложения, соответствующего этой грамматике, существует только одно дерево разбора, и нет неопределенности в выборе правил разбора. В однозначной грамматике каждая строка языка может быть однозначно связана с деревом разбора.

Примеры однозначных грамматик:

1. Грамматика арифметических выражений: Выражение -> Выражение + Терм | Выражение - Терм | Терм Терм -> Терм * Фактор | Терм / Фактор | Фактор Фактор -> Число | (Выражение)

   В этой грамматике для каждого арифметического выражения существует только одно дерево разбора, и порядок операций однозначно определяется структурой дерева разбора.
2. Грамматика простого языка программирования: Программа -> Объявление Последовательность*операторов Объявление -> int Идентификатор ; Последовательность*операторов -> Оператор ; Последовательность_операторов | Оператор ; Оператор -> Идентификатор = Выражение | if (Выражение) Оператор | while (Выражение) Оператор

   В этой грамматике каждое предложение программы имеет только одно возможное дерево разбора.

Примеры неоднозначных грамматик:

1. Грамматика арифметических выражений: Выражение -> Выражение + Выражение | Выражение - Выражение | Терм Терм -> Терм * Фактор | Терм / Фактор | Фактор Фактор -> Число | (Выражение)

   В этой грамматике существуют неоднозначности при разборе выражений, так как нет четкого правила о порядке операций.
2. Грамматика языка программирования с вложенными условными операторами: if_stmt -> if (Выражение) if_stmt | if (Выражение) Оператор Оператор -> ...

   В этой грамматике есть неоднозначности, когда вложенные условные операторы не имеют явной связи с соответствующим "else".

Неоднозначная грамматика может вызывать проблемы при синтаксическом анализе, поскольку существует несколько возможных деревьев разбора для одного предложения, что может привести к неоднозначности в семантике и трудностям при интерпретации предложения. Однозначные грамматики предпочтительны, так как они

обеспечивают однозначность в интерпретации предложений языка, основанного на этой грамматике.

3. Важные примеры однозначных грамматик: грамматика арифметических выражений (+, -, *, /) с учетом приоритетов операторов; оператор *if C then S__1 else S__2;* правильные скобочные выражения, цепочки из 0 и 1 с равным числом 0 и 1.

Пример однозначной грамматики для арифметических выражений с учетом приоритетов операторов (+, -, *, /):

```
Выражение -> Терм AddExpr AddExpr -> + Терм AddExpr | - Терм AddExpr | ε Терм -> Фактор MultExpr MultExpr -> * Фактор MultExpr | / Фактор MultExpr | ε Фактор -> Число | (Выражение)
```

Эта грамматика определяет, что умножение и деление имеют более высокий приоритет, чем сложение и вычитание. Таким образом, она гарантирует однозначность разбора арифметических выражений.

Пример однозначной грамматики для оператора if-then-else:

```
IfStmt -> if Condition then Stmt else Stmt Condition -> Expression Stmt -> Statement Expression -> ...
```

Эта грамматика однозначно определяет структуру оператора if-then-else и гарантирует, что для каждого if-then-else существует только одно возможное дерево разбора.

Пример однозначной грамматики для правильных скобочных выражений:

```
Выражение -> (Выражение) | Выражение Выражение | ε
```

Эта грамматика определяет правила для построения правильных скобочных выражений, где каждой открывающей скобке соответствует закрывающая скобка. Она гарантирует, что каждое скобочное выражение имеет только одно возможное дерево разбора.

Пример однозначной грамматики для цепочек из 0 и 1 с равным числом 0 и 1:

```
S -> 0S1 | 1S0 | ε
```

Эта грамматика определяет правила для построения цепочек из 0 и 1 с равным количеством 0 и 1. Она гарантирует, что каждая цепочка имеет только одно возможное дерево разбора.

4. Схема трансляции арифметических выражений в постфиксную польскую запись (ПОЛИЗ).

Схема трансляции арифметических выражений в постфиксную польскую запись (ПОЛИЗ) может быть реализована с использованием стека. Ниже приведена общая схема для такой трансляции:

1. Создать пустой стек операторов.
2. Создать пустой стек для записи постфиксной польской записи.
3. Проход по каждому символу входного арифметического выражения (слева направо).
    - Если символ является операндом (числом или переменной), добавить его в стек постфиксной записи.
    - Если символ является открывающей скобкой "(", поместить его в стек операторов.
    - Если символ является оператором:
        - Пока верхний элемент стека операторов имеет больший или равный приоритет и не является открывающей скобкой "(":
            - Перенести верхний элемент стека операторов в стек постфиксной записи.
        - Поместить текущий оператор в стек операторов.
    - Если символ является закрывающей скобкой ")":
        - Пока верхний элемент стека операторов не является открывающей скобкой "(":
            - Перенести верхний элемент стека операторов в стек постфиксной записи.
        - Удалить открывающую скобку "(" из стека операторов (не добавлять в стек постфиксной записи).
4. Пока стек операторов не пуст:
    - Перенести верхний элемент стека операторов в стек постфиксной записи.
5. Вернуть содержимое стека постфиксной записи как результат.

Пример: Для арифметического выражения "3 + 4 * 2 / (1 - 5)" схема трансляции в постфиксную польскую запись будет выглядеть следующим образом:

Шаг 1: Стек операторов: пусто Стек постфиксной записи: пусто

Шаг 2: Стек операторов: пусто Стек постфиксной записи: 3

Шаг 3: Стек операторов: + Стек постфиксной записи: 3

Шаг 4: Стек операторов: + Стек постфиксной записи: 3 4

Шаг 5: Стек операторов: + Стек постфиксной записи: 3 4 2

Шаг 6: Стек операторов: + * Стек постфиксной записи: 3 4 2

Шаг 7: Стек операторов: + *Стек постфиксной записи: 3 4 2*

Шаг 8: Стек операторов: + / Стек постфиксной записи: 3 4 2 *

Шаг 9: Стек операторов: + / Стек постфиксной записи: 3 4 2 * 1

Шаг 10: Стек операторов: + / Стек постфиксной записи: 3 4 2 * 1 5

Шаг 11: Стек операторов: + / Стек постфиксной записи: 3 4 2 * 1 5 -

Шаг 12: Стек операторов: + Стек постфиксной записи: 3 4 2 * 1 5 - /

Шаг 13: Стек операторов: пусто Стек постфиксной записи: 3 4 2 * 1 5 - / +

Таким образом, результатом трансляции будет "3 4 2 * 1 5 - / +", что является эквивалентом исходного арифметического выражения в постфиксной польской записи.

5. Нисходящий синтаксический анализ методом рекурсивного спуска. Определение и примеры. Функция FIRST(1). Ограничения на грамматику для применимости разбора цепочек этой грамматики данным методом.

Нисходящий синтаксический анализ методом рекурсивного спуска (Recursive Descent Parsing) - это один из методов синтаксического анализа, который основан на рекурсивных процедурах (функциях). Каждое правило грамматики представляется в виде отдельной функции, которая рекурсивно вызывает другие функции в соответствии с правилами разбора. Этот метод обычно используется для грамматик типа LL(k), где L означает слева-направо (left-to-right), а K - количество символов вперед, которые используются для принятия решения о правиле разбора.

Пример грамматики и соответствующих функций для нисходящего синтаксического анализа методом рекурсивного спуска:

Грамматика:

```
S -> E E -> T + E | T T -> F * T | F F -> (E) | id
```

Функции:

```
def parse_S():
    parse_E()

def parse_E():
    parse_T()
    if next_token() == '+':
        match('+')
        parse_E()
```

```
def parse_T():
    parse_F()
    if next_token() == '*':
        match('*')
        parse_T()

def parse_F():
    if next_token() == '(':
        match('(')
        parse_E()
        match(')')
    else:
        match('id')
```

В данном примере каждая функция соответствует правилу грамматики и выполняет разбор соответствующего нетерминала. Они вызывают другие функции, чтобы продолжить разбор правил грамматики.

Функция FIRST(1) - это функция, которая определяет множество терминалов, которые могут появиться в начале строки, порожденной данной грамматикой. Она используется в нисходящем синтаксическом анализе для определения следующего символа, который ожидается после текущего символа. Множество FIRST(1) для нетерминала A обычно включает первый символ каждой строки, порожденной A.

Ограничения на грамматику для применимости разбора цепочек методом рекурсивного спуска:

1. Грамматика должна быть однозначной, т.е. для каждого нетерминала не может быть неоднозначности в выборе правила разбора.
2. Грамматика должна быть LL(k), где L означает слева-направо (left-to-right), а K - количество символов вперед, которые используются для принятия решения о правиле разбора. Обычно используется LL(1), то есть решение о правиле разбора принимается на основе следующего символа во входной цепочке.
3. Грамматика не должна содержать левой рекурсии, т.е. рекурсивные правила должны быть праворекурсивными.
4. Грамматика должна быть возможной для представления в виде функций, где каждое правило грамматики соответствует отдельной функции, и функции должны быть рекурсивными.
5. Грамматика должна быть конечной и совместной, т.е. все символы должны быть различимыми и каждая входная цепочка должна иметь единственное синтаксическое дерево разбора.

Если эти ограничения выполняются, то метод рекурсивного спуска может быть успешно применен для разбора цепочек данной грамматики.

6. Регулярные выражения и регулярные множества. Приоритеты регулярных операторов (итерация, конкатенация, объединение). Некоторые примеры регулярных выражений: все цепочки из нулей единиц, заканчивающиеся на символ, который встречался в ней раньше; беззнаковое десятичное целое число, незначащие нули не допускаются; все цепочки из нулей единиц, в которых четное число нулей и четное число единиц.

Регулярные выражения - это формальный язык для описания и поиска шаблонов в строках. Они состоят из символов и операторов, которые определяют шаблон или множество строк.

Регулярные множества - это множества строк, которые могут быть сгенерированы или распознаны с использованием регулярных выражений. Они могут включать цепочки символов, соответствующие заданному шаблону.

Приоритеты регулярных операторов:

1. Итерация (знак "*", обозначает повторение нуля или более раз).
2. Конкатенация (объединение двух регулярных выражений или символов).
3. Объединение (знак "|", обозначает альтернативу между двумя регулярными выражениями или символами).

Примеры регулярных выражений:

1. Все цепочки из нулей и единиц, заканчивающиеся на символ, который встречался в ней раньше:

   ```
   (0*10*1)* | (1*01*0)*
   ```

   Это выражение описывает множество всех цепочек из нулей и единиц, в которых последний символ повторяется.
2. Беззнаковое десятичное целое число, незначащие нули не допускаются:

   ```
   [1-9][0-9]*
   ```

   Это выражение описывает множество всех беззнаковых десятичных целых чисел, в которых первая цифра не равна нулю и последующие цифры могут быть любыми цифрами от 0 до 9.
3. Все цепочки из нулей и единиц, в которых четное число нулей и четное число единиц:

   ```
   (00|11)*(0|1)
   ```

Это выражение описывает множество всех цепочек из нулей и единиц, в которых количество нулей и количество единиц являются четными числами.

Каждое из этих регулярных выражений может быть использовано для поиска соответствующих шаблонов в строках или для определения, принадлежит ли строка к заданному регулярному множеству.

7. Недетерминированный конечный автомат (НКА): определение, примеры. Алгоритм Томпсона для преобразования регулярного выражения в НКА.

Недетерминированный конечный автомат (НКА) - это математическая модель, используемая для описания и распознавания регулярных языков. Он состоит из конечного числа состояний и переходов между состояниями по входным символам. В отличие от детерминированного конечного автомата (ДКА), НКА может иметь несколько переходов для одного входного символа и может находиться в нескольких состояниях одновременно.

Примеры НКА:

1. НКА для распознавания языка, состоящего из строки "ab":
   a b
   → q0 -------→ q1 -|
   ↑ | |
   └──────────────┘ ↓
   ε q2
   НКА имеет три состояния q0, q1 и q2. Переход от q0 до q1 возможен по символу "a", и переход от q1 до q2 возможен по символу "b". Строка "ab" будет принята НКА.
2. НКА для распознавания языка, состоящего из строк, содержащих две последовательные нули:
   0 0
   → q0 -------→ q1 -|
   ↑ | |
   └──────────────┘ ↓
   ε q2
   НКА имеет три состояния q0, q1 и q2. Переход от q0 до q1 и от q1 до q2 возможен по символу "0". Любая строка, содержащая две последовательные нули, будет принята НКА.

Алгоритм Томпсона (Thompson's algorithm) используется для преобразования регулярного выражения в эквивалентный НКА. Он состоит из следующих шагов:

1. Для каждого символа регулярного выражения создается начальное и конечное состояния НКА.

2. После этого выполняются операции, соответствующие операторам регулярных выражений:
    - Конкатенация: Присоединение конечного состояния первого операнда к начальному состоянию второго операнда.
    - Объединение: Создание нового начального и конечного состояний, а затем добавление эпсилон-переходов из нового начального состояния в начальные состояния обоих операндов, а также из конечных состояний обоих операндов в новое конечное состояние.
    - Итерация: Создание нового начального и конечного состояний, а затем добавление эпсилон-переходов из нового начального состояния в начальное состояние операнда, а также из конечного состояния операнда в новое конечное состояние, и эпсилон-переход из нового начального состояния в новое конечное состояние.
3. После выполнения всех операций регулярного выражения, начальное состояние НКА становится начальным состоянием результата, а конечное состояние НКА становится конечным состоянием результата.

Таким образом, алгоритм Томпсона позволяет построить эквивалентный НКА для данного регулярного выражения.

8. Детерминированный конечный автомат (ДКА): определение, примеры. Алгоритм построения ДКА из НКА.

Детерминированный конечный автомат (ДКА) - это математическая модель, используемая для описания и распознавания регулярных языков. В отличие от недетерминированного конечного автомата (НКА), ДКА имеет одно определенное состояние для каждой комбинации текущего состояния и входного символа.

Примеры ДКА:

1. ДКА для распознавания языка, состоящего из строки "ab":
   a b
   → q0 --------→ q1 -|
   ↑ | |
   └──────────────┘ ↓
   ε q2
   ДКА имеет три состояния q0, q1 и q2, и каждый переход происходит по определенному символу. При обработке строки "ab" ДКА последовательно переходит из состояния в состояние и принимает строку.
2. ДКА для распознавания языка, состоящего из строк, содержащих две последовательные нули:
   0 1
   → q0 ----------→q1 -|
   ↑ | |

└──────────┘ ↓

0 q2

ДКА имеет три состояния q0, q1 и q2, и каждый переход происходит по определенному символу. Любая строка, содержащая две последовательные нули, будет принята ДКА.

Алгоритм построения ДКА из НКА (Subset Construction или алгоритм подмножества) состоит из следующих шагов:

1. Начальное состояние ДКА - это ε-замыкание (ε-closure) начального состояния НКА.
2. Пока есть новые состояния для обработки:
   - Для каждого символа входного алфавита:
     - Выполнить ε-переходы из текущего состояния НКА.
     - Получить новое состояние путем перемещения из текущего состояния по символу.
     - Выполнить ε-замыкание для нового состояния.
     - Если новое состояние не является пустым и не содержится в ДКА, добавить его в ДКА как новое состояние для обработки.
     - Добавить переход из текущего состояния ДКА в новое состояние ДКА по символу.
3. Определить конечные состояния ДКА как состояния, содержащие конечные состояния НКА.

Таким образом, алгоритм подмножества позволяет построить эквивалентный ДКА для данного НКА.

9. Алгоритм преобразования ДКА в эквивалентный ДКА с минимальным количеством состояний.

Алгоритм преобразования ДКА в эквивалентный ДКА с минимальным количеством состояний называется алгоритмом минимизации ДКА (DFA minimization). Этот алгоритм позволяет упростить ДКА, объединяя эквивалентные состояния.

Алгоритм минимизации ДКА состоит из следующих шагов:

1. Изначально разделите состояния ДКА на две группы: конечные и неконечные состояния.
2. Создайте начальное разбиение состояний, где конечные и неконечные состояния разделены в разные группы.
3. Повторяйте следующие шаги до тех пор, пока разбиение не стабилизируется:
   - Для каждой группы состояний:
     - Разделите эту группу на подгруппы, используя эквивалентность состояний на основе переходов по каждому символу входного алфавита.
   - Если разбиение изменилось, обновите текущее разбиение состояний.

4. Постройте новый ДКА, где каждая подгруппа из разбиения представляет одно объединенное состояние.
5. Установите начальное состояние нового ДКА в группу, содержащую изначальное начальное состояние.
6. Установите конечные состояния нового ДКА в группы, содержащие изначальные конечные состояния.

После выполнения алгоритма минимизации ДКА, полученный ДКА будет иметь минимальное количество состояний, при сохранении эквивалентности исходного ДКА. Это позволяет упростить автомат и улучшить его эффективность при распознавании языка.

Важно отметить, что алгоритм минимизации ДКА имеет экспоненциальную сложность, и для некоторых ДКА может потребоваться значительное время для выполнения. Однако, для многих практических случаев алгоритм обладает достаточной эффективностью.

10. Построение диаграммы переходов конечного автомата по заданной автоматной грамматике.

Построение диаграммы переходов конечного автомата (или конечного автомата) по заданной автоматной грамматике можно выполнить следующим образом:

1. Задайте множество состояний автомата, включая начальное состояние и конечные состояния.
2. Определите входной алфавит, то есть множество символов, которые могут быть прочитаны автоматом.
3. Задайте переходы между состояниями на основе правил автоматной грамматики. Каждое правило должно указывать текущее состояние, символ входного алфавита и следующее состояние, в которое автомат должен перейти.
4. Постройте диаграмму переходов, где каждое состояние представлено узлом (обычно кругом или прямоугольником) и переходы между состояниями представлены стрелками, подписанными символами входного алфавита.
5. Отметьте начальное состояние, например, с помощью стрелки, указывающей на него или подписи "начальное состояние".
6. Отметьте конечные состояния, например, двойными кругами или подписью "конечное состояние".

Пример:

Пусть у нас есть автоматная грамматика с правилами: S -> aA A -> b

Мы можем построить диаграмму переходов следующим образом:

1. Множество состояний: {S, A}

2. Входной алфавит: {a, b}
3. Переходы:
   - Из состояния S по символу 'a' переходим в состояние A.
   - Из состояния A по символу 'b' переходим в состояние A.
4. Диаграмма переходов:
   a
   S ----------> A
   b
5. Начальное состояние: S
6. Конечное состояние: A

В этом примере у нас два состояния - S и A. Стрелка с символом 'a' указывает на переход от состояния S к состоянию A. Стрелка с символом 'b' указывает на переход от состояния A к тому же состоянию A, что является конечным состоянием.

Таким образом, диаграмма переходов конечного автомата позволяет наглядно представить переходы между состояниями в соответствии с автоматной грамматикой.

11. Лемма о разрастании для регулярных множеств: формулировка, применение леммы для доказательства нерегулярности множеств: (1) {0n1n | n>=1}; (2) {все строки из 0 и 1, в которых число «единиц» = числу «нулей»}.

Лемма о разрастании (Pumping Lemma) для регулярных множеств - это важный инструмент, который позволяет доказать нерегулярность некоторых языков. Формулировка леммы о разрастании следующая:

Пусть L - регулярное множество. Тогда существует некоторая константа p (длина разрастающейся строки), такая что для любой строки w ∈ L, |w| ≥ p, можно представить строку w в виде w = xyz, где выполняются следующие условия:

1. |xy| ≤ p.
2. |y| > 0.
3. Для любого натурального числа i ≥ 0, строка xy^iz также принадлежит L.

Применение леммы о разрастании для доказательства нерегулярности множеств заключается в поиске разрастающейся строки, которая нарушает одно из условий леммы. Рассмотрим применение леммы о разрастании для двух множеств:

1. Множество {0^n1^n | n ≥ 1}: Предположим, что это множество является регулярным. Возьмем разрастающуюся строку w = 0^p1^p, где p - константа из леммы. Разложим строку w на xyz, соблюдая условия леммы. В этом случае, y может содержать только символы '0'. При повторном накачивании y (i > 1), количество символов '0' в начале строки будет больше количества символов '1' в

конце строки, что противоречит условию языка. Следовательно, множество {0^n1^n | n ≥ 1} не является регулярным.
2. Множество {все строки из 0 и 1, в которых число "единиц" = числу "нулей"}: Предположим, что это множество является регулярным. Возьмем разрастающуюся строку w = 0^p1^p0^p, где p - константа из леммы. Разложим строку w на xyz, соблюдая условия леммы. В этом случае, y может содержать только символы '0'. При повторном накачивании y (i > 1), количество символов '0' в начале строки будет превышать количество символов '1' и '0' в конце строки, что противоречит условию языка. Следовательно, множество {все строки из 0 и 1, в которых число "единиц" = числу "нулей"} не является регулярным.

Таким образом, лемма о разрастании позволяет доказывать нерегулярность некоторых множеств, путем нахождения разрастающихся строк, которые не могут быть распознаны регулярным автоматом.

12. Устранение непосредственной левой рекурсии в КС-грамматике. Левая факторизация.

Устранение непосредственной левой рекурсии и левая факторизация - это две техники, которые часто применяются для модификации контекстно-свободных грамматик (КС-грамматик) с целью улучшения их использования в алгоритмах синтаксического анализа. Давайте рассмотрим каждую из них подробнее:

1. Устранение непосредственной левой рекурсии: Непосредственная левая рекурсия возникает в грамматике, когда правило имеет форму A -> Aα, где A - нетерминал, α - последовательность символов, включающая терминалы и/или другие нетерминалы. Непосредственная левая рекурсия может вызывать бесконечные циклы при попытке разбора цепочек.

   Для устранения непосредственной левой рекурсии можно использовать следующий подход:

   - Представьте правила с непосредственной левой рекурсией в виде: A -> αA' | β A' -> αA' | β
   - Создайте новый нетерминал A', который будет использоваться для обозначения продолжения строки, соответствующей A.
   - Замените все правила с непосредственной левой рекурсией на новые правила, исключая непосредственную левую рекурсию. В новых правилах нетерминал A не должен непосредственно раскрываться в самого себя.

   Пример устранения непосредственной левой рекурсии: Рассмотрим грамматику с непосредственной левой рекурсией: A -> Aα | β Для устранения рекурсии можно заменить правила следующим образом: A -> βA' A' -> αA' | ε
2. Левая факторизация: Левая факторизация используется для устранения общих префиксов в правилах грамматики. Общий префикс возникает, когда два или

более правила имеют одинаковую последовательность символов в начале.

Для выполнения левой факторизации можно использовать следующий подход:

- Выделите общий префикс в правилах и создайте новый нетерминал для представления этого префикса.
- Замените правила с общим префиксом на новые правила, в которых общий префикс заменяется на новый нетерминал.
- Добавьте продолжения для каждого правила, в которых остаток правила добавляется после нового нетерминала.

Пример левой факторизации: Рассмотрим грамматику с общим префиксом: A -> αβγ | αδ Для выполнения левой факторизации можно заменить правила следующим образом: A -> αA' A' -> βγ | δ

Обе техники, устранение непосредственной левой рекурсии и левая факторизация, помогают преобразовать КС-грамматику для более эффективного синтаксического анализа и избежания проблем, связанных с рекурсивными вызовами и общими префиксами.

13. Функции FIRST(1) и FOLLOW(1). Построение таблицы предиктивного синтаксического анализа для LL(1) грамматик.

Функции FIRST(1) и FOLLOW(1) используются в синтаксическом анализе для анализа и построения таблицы предиктивного синтаксического анализа для LL(1) грамматик. Давайте рассмотрим каждую функцию и построение таблицы подробнее:

1. Функция FIRST(1): Функция FIRST(1) определяет множество терминалов, которые могут быть первыми символами строк, порождаемых из данного символа грамматики. Функция FIRST(1) определяется следующим образом:
   - Если X - терминал, то FIRST(1)(X) = {X}.
   - Если X - нетерминал, то FIRST(1)(X) содержит все терминалы, которые могут быть первыми символами строк, порождаемых из X, а также ε (пустая строка), если X может породить пустую строку.
2. Функция FOLLOW(1): Функция FOLLOW(1) определяет множество терминалов, которые могут следовать непосредственно за данным символом в строках, порождаемых из грамматики. Функция FOLLOW(1) определяется следующим образом:
   - FOLLOW(1)(S) содержит символ $ (конец строки), если S - стартовый символ грамматики.
   - Если A - нетерминал, и есть правило вида B -> αAβ, где α и β - последовательности символов (возможно, пустые), то все терминалы из FIRST(1)(β), кроме ε, добавляются в FOLLOW(1)(A).

- Если A - нетерминал, и есть правило вида B -> αA, где α - последовательность символов (возможно, пустая), то все терминалы из FOLLOW(1)(B) добавляются в FOLLOW(1)(A).

3. Построение таблицы предиктивного синтаксического анализа для LL(1) грамматик: Построение таблицы предиктивного синтаксического анализа включает следующие шаги:

   - Для каждого правила A -> α, где A - нетерминал, вычислите FIRST(1)(α) и добавьте все терминалы, кроме ε, в ячейки таблицы, соответствующие комбинациям (A, t), где t - терминал из FIRST(1)(α).
   - Если ε принадлежит FIRST(1)(α), для каждого терминала t из FOLLOW(1)(A), добавьте правило A -> ε в ячейку таблицы, соответствующую (A, t).
   - Если есть конфликты в таблице (несколько правил в одной ячейке), то грамматика не является LL(1).

   Таблица предиктивного синтаксического анализа позволяет предсказывать следующий шаг анализатора на основе текущего символа входной строки и верхнего символа стека. Каждая ячейка таблицы может содержать одно правило грамматики, которое должно быть применено для разбора.

Это общий подход к построению таблицы предиктивного синтаксического анализа для LL(1) грамматик. Он основывается на функциях FIRST(1) и FOLLOW(1), которые определяются для каждого символа грамматики.

14. Пример *не* LL(1) грамматики. Грамматика примера должна иметь бесконечное число различных цепочек

Вот пример грамматики, которая не является LL(1) из-за неоднозначности:

S -> aSb | ε

Эта грамматика имеет бесконечное число различных цепочек. Она порождает строки, состоящие из произвольного количества символов 'a', за которыми следует такое же количество символов 'b'. Таким образом, цепочки, порождаемые этой грамматикой, могут выглядеть следующим образом:

ε, ab, aabb, aaabbb, aaaabbbb, и так далее.

Однако, грамматика содержит неоднозначность, так как для любой цепочки из символов 'a' и 'b', существует несколько различных выводов. Например, для цепочки 'aabb', мы можем использовать первое правило и породить 'aSb' -> 'aaSbb' -> 'aabb', или использовать второе правило и породить 'ε'. Это приводит к неоднозначности разбора и делает грамматику не LL(1).

15. Отношения предшествования. Грамматики простого предшествования. Построение матрицы предшествования.

Отношения предшествования используются в анализе и построении таблицы операторного предшествования для алгоритма синтаксического анализа с использованием метода операторного предшествования (operator-precedence parsing). Они определяют порядок выполнения операторов в грамматике.

1. Отношения предшествования:
   - Отношение "меньше" (<): Если символ A может непосредственно следовать за символом B в правиле грамматики, то отношение B < A существует.
   - Отношение "больше" (>): Если символ A может непосредственно предшествовать символу B в правиле грамматики, то отношение A > B существует.
   - Отношение "равно" (=): Если символ A и символ B могут непосредственно следовать друг за другом в правиле грамматики, то отношение A = B существует.
2. Грамматики простого предшествования: Грамматики простого предшествования (simple precedence grammars) являются подклассом контекстно-свободных грамматик, в которых каждая грамматическая форма (правило) имеет одно из следующих свойств:
   - Нет двух соседних нетерминалов.
   - Нет двух соседних терминалов.
   - Нетерминал находится перед терминалом.
3. Построение матрицы предшествования: Матрица предшествования представляет собой квадратную матрицу, где строки и столбцы соответствуют символам грамматики (терминалы и нетерминалы), а элементы матрицы представляют отношения предшествования между символами. Для построения матрицы предшествования:

   - Инициализируйте все элементы матрицы пустыми значениями или некоторым значением по умолчанию.
   - Для каждого правила грамматики вида A -> αBβ, где B - символ грамматики, определите отношение B < a для каждого символа a из FIRST(β), и отношение a > B для каждого символа a из FOLLOW(B). Запишите соответствующие отношения в соответствующие элементы матрицы.
   - Если у вас есть правило грамматики вида A -> αB или A -> αBβ, где B - символ грамматики, и ε принадлежит FIRST(β), то запишите отношение B < a и B > a для каждого символа a из FOLLOW(A).
   - Если у вас есть правило грамматики вида A -> βB или A -> βBγ, где B - символ грамматики, и ε принадлежит FIRST(γ), то запишите отношение a < B и a > B для каждого символа a из FIRST(β).

Построенная матрица предшествования может быть использована для определения следующего шага в синтаксическом анализе, основанном на операторном предшествовании, на основе текущего символа входной строки и верхнего символа стека.

16. Алгоритм построения матрицы предшествования для грамматики *операторного* предшествования. Алгоритм разбора перенос-свертка для грамматики операторного предшествования. Остовная грамматика.

Алгоритм построения матрицы предшествования для грамматики операторного предшествования:

1. Инициализация: Создайте пустую матрицу предшествования размером n x n, где n - количество символов грамматики (терминалов и нетерминалов). Заполните все элементы матрицы пустыми значениями или некоторым значением по умолчанию.
2. Для каждого правила грамматики вида A -> αBβ, где B - символ грамматики:
   - Для каждого символа a из FIRST(β), запишите отношение B < a в соответствующую ячейку матрицы.
   - Если ε принадлежит FIRST(β):
     - Для каждого символа a из FOLLOW(A), запишите отношение B < a в соответствующую ячейку матрицы.
     - Для каждого символа a из FOLLOW(B), запишите отношение a > B в соответствующую ячейку матрицы.
3. Для каждого правила грамматики вида A -> βB или A -> βBγ, где B - символ грамматики, и ε принадлежит FIRST(γ):
   - Для каждого символа a из FIRST(β), запишите отношение a < B в соответствующую ячейку матрицы.
   - Для каждого символа a из FOLLOW(B), запишите отношение a > B в соответствующую ячейку матрицы.
4. Алгоритм построения матрицы предшествования завершен.

Алгоритм разбора перенос-свертка для грамматики операторного предшествования:

1. Инициализация: Создайте стек и поместите на него начальный символ грамматики и символ конца строки ($).
2. Пока входная строка не исчерпана:
   - Просмотрите верхний символ стека (top) и текущий символ входной строки (current).
   - Если top < current, сделайте перенос (shift) и поместите current в стек, считайте следующий символ из входной строки.
   - Если top > current, сделайте свертку (reduce) с использованием правила грамматики, указанного в соответствующей ячейке матрицы

предшествования, и удалим из стека соответствующее количество символов.
    - Если top = current, сделайте перенос и свертку (shift-reduce) по правилу грамматики, указанному в соответствующей ячейке матрицы предшествования.
    - Если ни одно из вышеуказанных условий не выполняется, возникает ошибка синтаксического анализа.
3. Если входная строка и стек пусты, алгоритм разбора завершен успешно. В противном случае возникает ошибка синтаксического анализа.

Остовная грамматика (skeleton grammar) - это грамматика, созданная путем удаления некоторых правил грамматики или их модификации с целью упрощения или обобщения грамматики. Она может использоваться, например, для построения таблицы предшествования в алгоритме операторного предшествования или для устранения левой рекурсии и левой факторизации в КС-грамматике. Остовная грамматика обычно сохраняет основную структуру и свойства исходной грамматики, но может иметь некоторые упрощения или изменения, чтобы облегчить анализ или обработку.

17. LR(0)-грамматика. Алгоритм построения таблицы разбора.

LR(0)-грамматика является классом контекстно-свободных грамматик, для которых используется алгоритм LR(0)-анализа (или канонический LR(0)-анализ). LR(0)-анализ является одним из методов синтаксического анализа, который строит таблицу разбора для грамматики.

Алгоритм построения таблицы разбора для LR(0)-грамматики:

1. Построение наборов пунктов (item sets):
    - Начните с начального пункта, содержащего правило грамматики с маркером в начале (например, [S' -> ·S]).
    - Для каждого пункта в наборе:
        - Если после маркера находится нетерминал, добавьте все правила грамматики для этого нетерминала с маркером в начале в текущий набор.
        - Если после маркера находится терминал, перейдите к следующему пункту.
2. Замыкание (closure) наборов пунктов:
    - Для каждого нетерминала A в каждом пункте [A -> α·Bβ]:
        - Добавьте все правила грамматики для B с маркером в начале в текущий набор, если они еще не были добавлены.
3. Построение таблицы разбора:
    - Создайте таблицу разбора размером M x N, где M - количество наборов пунктов, N - количество терминалов и нетерминалов грамматики.

- Для каждого пункта [A -> α·aβ] в наборе i и каждого терминала a:
  - Если a - терминал, добавьте действие "сдвиг" (shift) в таблицу разбора в ячейку [i, a], указывая, что следующим шагом должно быть считывание символа a и переход к набору, соответствующему пункту [A -> αa·β].
- Для каждого пункта [A -> α·] в наборе i и каждого символа a:
  - Если A - стартовый символ грамматики, добавьте действие "принятие" (accept) в таблицу разбора в ячейку [i, $], указывая, что разбор завершен успешно.
  - Иначе, для каждого нетерминала B в FOLLOW(A), добавьте действие "свертка" (reduce) в таблицу разбора в ячейку [i, b], указывая, что следующим шагом должна быть свертка по правилу [A -> α·] (где α - последовательность символов перед A).

4. Разрешение конфликтов:
   - Если в таблице разбора возникают конфликты, например, неоднозначности или пересечения действий, грамматика не является LR(0)-грамматикой и требует более мощного анализатора, такого как SLR(1), LALR(1) или LR(1).

Алгоритм построения таблицы разбора для LR(0)-грамматики завершен. Таблица разбора может использоваться в процессе синтаксического анализа методом LR(0)-анализа для разбора входной строки и определения допустимости грамматики и порядка действий разбора.

18. SLR-грамматика. Алгоритм построения таблицы разбора.

SLR-грамматика (Simple LR-грамматика) является классом контекстно-свободных грамматик, для которых используется алгоритм SLR-анализа (Simple LR-анализа). SLR-анализ является одним из методов синтаксического анализа, который строит таблицу разбора для грамматики.

Алгоритм построения таблицы разбора для SLR-грамматики:

1. Построение наборов пунктов (item sets):
   - Начните с начального пункта, содержащего правило грамматики с маркером в начале (например, [S' -> ·S]).
   - Для каждого пункта в наборе:
     - Если после маркера находится нетерминал, добавьте все правила грамматики для этого нетерминала с маркером в начале в текущий набор.
     - Если после маркера находится терминал, перейдите к следующему пункту.
2. Замыкание (closure) наборов пунктов:
   - Для каждого нетерминала A в каждом пункте [A -> α·Bβ]:
     - Добавьте все правила грамматики для B с маркером в начале в текущий набор, если они еще не были добавлены.

3. Построение таблицы разбора:
    - Создайте таблицу разбора размером M x N, где M - количество наборов пунктов, N - количество терминалов и нетерминалов грамматики.
    - Для каждого пункта [A -> α·aβ] в наборе i и каждого терминала a:
        - Если a - терминал, добавьте действие "сдвиг" (shift) в таблицу разбора в ячейку [i, a], указывая, что следующим шагом должно быть считывание символа a и переход к набору, соответствующему пункту [A -> αa·β].
    - Для каждого пункта [A -> α·] в наборе i и каждого символа b в FOLLOW(A):
        - Добавьте действие "свертка" (reduce) в таблицу разбора в ячейку [i, b], указывая, что следующим шагом должна быть свертка по правилу A -> α.
    - Если для некоторого пункта [A -> α·] и символа b в таблице разбора уже существует запись, неоднозначность, добавьте конфликт в таблицу разбора.
4. Разрешение конфликтов:

```
-   Если в таблице разбора возникают конфликты, например,
неоднозначности или пересечения действий, грамматика не является SLR-
грамматикой и может потребоваться использование более мощных
анализаторов, таких как LALR(1) или LR(1).
```

Алгоритм построения таблицы разбора для SLR-грамматики завершен. Таблица разбора может использоваться в процессе синтаксического анализа методом SLR-анализа для разбора входной строки и определения допустимости грамматики и порядка действий разбора.