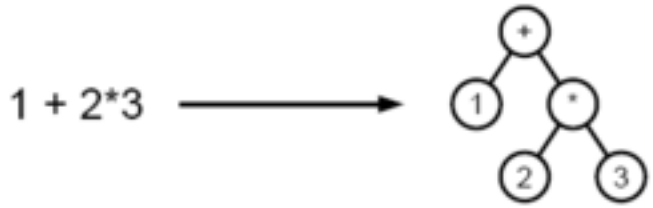
В информатике, синтаксический анализ (парсинг) — это процесс сопоставления линейной последовательности лексем (слов, токенов) языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом. Синтаксический анализатор (парсер) — это программа или часть программы, выполняющая синтаксический анализ.

При парсинге исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки. Пример разбора выражения в дерево:



Всё что угодно, имеющее «синтаксис», поддается автоматическому анализу.

- Языки программирования — разбор исходного кода языков программирования, в процессе трансляции (компиляции или интерпретации).
- Структурированные данные — данные, языки их описания, оформления и т. д. Например, XML, HTML, CSS, ini-файлы, специализированные конфигурационные файлы и т. п.
- SQL-запросы (DSL-язык)
- математические выражения
- регулярные выражения (которые, в свою очередь, могут использоваться для автоматизации лексического анализа)
- формальные грамматики
- Лингвистика — человеческие языки. Например, машинный перевод и другие генераторы текстов.

Восстановление после ошибок

Если компилятор будет исеть дело исключительно с корректными программами, его разработка и реализация существенно упростится. Однако, от компилятора ожидается, что он будет помогать программисту обнаруживать и устранять ошибки, которые неизбежно содержатся в программах несмотря на огромные усилия даже самых квалифицированных программистов. Примечательно, что хотя ошибки — явление чрезвычайно распространенное, лишь в нескольких языках вопрос обработки ошибок рассматривался на фазе проектирования языка. Большинство спецификаций языков программирования, тем не менее, не определяет реакции компилятора на ошибки — этот впорос отдается на откуп разработчикам компилятора. Однако, планирование системы обработки ошибок с самого начала работы над компилятором может как упростить его структуру, так и улучшить его реакцию на ошибки.

Как мы с вами говорили на прошлом занятии, ошибки в программе могут быть на

самых разных уровнях:

- *Лексические* ошибки включают неверно записанные идентификаторы, ключевые слова и операторы. Например, это может быть отсутствие кавычек вокруг текста, являющегося строкой, или опечатки в именах переменных.
- *Синтаксические* ошибки включают неверно поставленные точки с запятой или лишние или недостающие фигурные скобки. В качестве еще одного примера в C++ или Java синтаксической ошибкой является конструкция case без охватывающего switch.
- *Семантические* ошибки включают в себя несоответствие типов операторов и их операндов. В качестве примера можно рассмотреть оператор return, возвращающий какое-то определенное значение в функции, возвращающей void.
- *Логические* ошибки могут быть совершенно любыми — от неверных решений программиста до использования в программах на C++ оператора присваивания = вместо оператора сравнения ==. Такая программа может быть синтаксически корректной, но делать совсем не то, чего от нее хотел программист.

Точность современных методов разбора позволяет очень эффективно выявлять синтаксические ошибки в программе. Некоторые методы синтаксического анализа, такие как LL и LR (будем разбирать позже), обнаруживают ошибки на самых ранних стадиях, т.е. когда разбор потока токенов от лексического анализатора в соответствии с грамматикой языка становится невозможен.

Еще одна причина, по которой делается упор на восстановление после ошибки в процессе синтаксического анализа — многие ошибки, чем бы они ни были вызваны, оказываются синтаксическими и выявляются при дальнейшей невозможности синтаксического анализа. Эффективно обнаруживаются и некоторые семантические ошибки, такие, как несоответствия типов. Однако, в общем случае точное определение семантических и логических ошибок во время компиляции является крайне трудной задачей.

Простейший способ реагирования на некорректную входную цепочку лексем — завершить синтаксический анализ и вывести сообщение об ошибке. Однако часто оказывается полезным найти за одну попытку синтаксического анализа как можно больше ошибок. Именно так ведут себя трансляторы большинства распространённых языков программирования.

Таким образом перед обработчиком ошибок синтаксического анализатора стоят следующие задачи:

- он должен ясно и точно сообщать о наличии ошибок;
- он должен обеспечивать быстрое восстановление после ошибки, чтобы продолжать поиск других ошибок;
- он не должен существенно замедлять обработку корректной входной цепочки.

Рассмотрим наиболее известные стратегии восстановления после ошибок.

Восстановление в режиме паники

При обнаружении ошибки синтаксический анализатор пропускает входные лексемы по одной, пока не будет найдена одна из специально определенного множества

синхронизирующих лексем. Обычно такими лексемами являются разделители, например, символы “;”, “)” или “}”. Набор синхронизирующих лексем должен определять разработчик анализируемого языка. При такой стратегии восстановления может оказаться, что значительное количество символов будут пропущены без проверки на наличие дополнительных ошибок. Данная стратегия восстановления наиболее проста в реализации.

Восстановление на уровне фразы

Иногда при обнаружении ошибки синтаксический анализатор может выполнить локальную коррекцию входного потока так, чтобы это позволило ему продолжать работу. Например, перед точкой с запятой, отделяющей различные операторы в языке программирования, синтаксический анализатор может закрыть все ещё не закрытые круглые скобки. Это более сложный в проектировании и реализации способ, однако в некоторых ситуациях, он может работать значительно лучше восстановления в режиме паники. Естественно, данная стратегия бессильна, если настоящая ошибка произошла до точки обнаружения ошибки синтаксическим анализатором.

Продукции ошибок

Знание наиболее распространённых ошибок позволяет расширить грамматику языка продукциями, порождающими ошибочные конструкции. При срабатывании таких продукций регистрируется ошибка, но синтаксический анализатор продолжает работать в обычном режиме.

Формальные грамматики

Формальная грамматика или просто грамматика в теории формальных языков — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита.

- **Терминал** (терминальный символ) — объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»). В формальных языках, используемых на компьютере, в качестве терминалов обычно берут все или часть стандартных символов ASCII — латинские буквы, цифры и спец. символы.
- **Нетерминал** (нетерминальный символ) — объект, обозначающий какую-либо сущность языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

Чтобы задать грамматику, требуется задать алфавиты терминалов и нетерминалов, набор правил вывода, а также выделить в множестве нетерминалов начальный.

Итак, грамматика определяется следующими характеристиками:

- Σ —алфавит терминальных символов
- N — алфавит нетерминальных символов
- P — набор правил (продукций грамматики)
- S — стартовый (начальный) символ из набора нетерминалов.

Продукции грамматики определяют способ, которым терминалы и нетерминалы могут объединяться для создания строк. Каждая продукция состоит из следующих частей:

- Нетерминал, именуемый заголовком (или левой частью) продукции. Это продукция определяет некоторые из строк, обозначаемые заголовком.
- Символ ->. Иногда вместо стрелки используется ::=.
- Тело (или правая часть), состоящее из нуля или некоторого количества терминалов и нетерминалов. Эти компоненты тела описывают один из способов, которым могут быть построены строки нетерминала в заголовке.

Выводом называется последовательность строк, состоящих из терминалов и нетерминалов, где первой идет строка, состоящая из одного стартового нетерминала, а каждая последующая строка получена из предыдущей путем замены некоторой подстроки по одному (любому) из правил. Конечной строкой является строка, полностью состоящая из терминалов, и следовательно являющаяся словом языка.

Словами языка, заданного грамматикой, являются все последовательности терминалов, выводимые (порождаемые) из начального нетерминала по правилам вывода. Существование вывода для некоторого слова является критерием его принадлежности к языку, определяемому данной грамматикой.

Пример

Рассмотрим грамматику для языка, использующегося для описания выражений, содержащих слагаемые и сомножители. Такой язык будет описываться следующим набором правил:

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Здесь E представляет собой выражение, состоящее из слагаемых, разделенных знаками +, T представляет слагаемые, разделенные знаками *, а F представляет сомножители, которые могут быть либо выражениями в скобках, либо просто идентификаторами.

Так вот, терминальнами символами являются следующие символы: {id, +, *, (,)}.

Нетерминальными символами являются E, T, F, причем E — старторвый символ грамматики.

Типы грамматик

По иерархии Хомского, грамматики делятся на 4 типа, каждый последующий является более ограниченным подмножеством предыдущего (но и легче поддающимся анализу):

- тип 0. неограниченные грамматики — возможны любые правила. Этот класс грамматик представляет теоретический интерес, но практически не применяется как таковой
- тип 1. контекстно-зависимые грамматики — левая часть может содержать один

нетерминал, окруженный «контекстом» (последовательности символов, в том же виде присутствующие в правой части); сам нетерминал заменяется непустой последовательностью символов в правой части.

- тип 2. контекстно-свободные грамматики — левая часть состоит из одного нетерминала.
- тип 3. регулярные грамматики — более простые, эквивалентны конечным автоматам и регулярным выражениям

Нам будут интересны КС грамматики. Смысл термина «контекстно-свободная» заключается в том, что возможность применить продукцию к нетерминалу, в отличие от общего случая, не зависит от контекста этого нетерминала. КС-грамматики находят большое применение в информатике. Ими задаётся грамматическая структура большинства языков программирования, структурированных данных и т.д.

Примеры КС-грамматик и соответствующих им КС-языков:

Вложенные скобки

- Терминалы: '(' и ')';
- нетерминал: S;
- продукции: S→(S), S→ε;
- начальный нетерминал — S.

Этой грамматикой задаётся язык вложенных скобок $\{ (_n)_n \mid n \geq 0 \}$.

Не все языки могут быть заданы КС-грамматикой. Так, язык
$\mathbf{\{ a_n b_n c_n \mid n \geq 1 \}}$ не является контекстно-свободным.

Построение вывода

Построение дерева разбора можно сделать совершенно точным при помощи порождений, рассматривая каждую продукцию, как правило для переписывания. Начиная со стартового символа, на каждом шаге переписывания нетерминал заменяется телом одной из его продукций.

Рассмотрим, например, следующую грамматику:

```
E -> E + E | E * E | -E | (E) | id
```

Продукция E -> -E означает, что если E обозначает выражение, то -E также должно обозначать выражение. Замена одного E на -E может быть описана записью “E => -E”. Она читается как "E порождает -E".

Продукция E -> (E) может быть применена для замены любого экземпляра E в любой строке символов грамматики на (E), например E * E => (E) * E или E * E => E * (E). Можно взять один нетерминал E и многократно применять продукции в произвольном порядке для получения последовательности замещений, например, E => -E => -(E) => -(id).

Такая последовательность замен называется выводом (derivation) или порождением -(id) из E. Это порождение доказывает, что строка -(id) является конкретным примером выражения.

Чтобы дать общее определение порождения, рассмотрим нетерминал A в средине последовательности грамматических символов, как в случае aAb, где a и b — произвольные строки грамматических символов. Предположим, что A -> c является продукцией. Тогда мы записываем aAb => acb. Символ => означает "порождение за один шаг". Если последовательность порождений a_1 => a_2 => … => a_n переписывает a_1, заменяя его a_n, мы говорим, что a_1 порождает a_n.

Строка “-(id + id)” является предложением рассматриваемой грамматики ввиду наличия следующего порождения:

```
E => -E => -(E) => -(E + E) => -(id + E) => -(id + id)        (*)
```

На каждом шаге порождения осуществляется два выбора — мы должны выбрать заменяемый нетерминал, а затем продукцию, у которой данный нетерминал является заголовком. Например, приведенное ниже альтернативное порождения для “-(id + id)” отличается от порождения выше последними двумя шагами:

```
E => -E => -(E) => -(E + E) => -(E + id) => -(id + id)       (**)
```

Каждый нетерминал заменяется тем же телом, что и ранее, но в другом порядке.

Чтобы понять, как работают синтаксические анализаторы, рассмотрим порождения, в которых замещаемый нетерминал на каждом шаге выбирается следующим образом.

1. В *левых* (leftmost) порождениях в каждом предложении всегда выбирается крайний слева нетерминал. Если $\alpha \Rightarrow \beta$ является шагом, на котором замещается крайний слева нетерминал α, то это записывается как $\alpha \underset{lm}{\Rightarrow} \beta$.

2. В *правых* (rightmost) порождениях в каждом предложении всегда выбирается крайний справа нетерминал; в этом случае мы пишем $\alpha \underset{rm}{\Rightarrow} \beta$.

Так вот, порождение (*) — левое, а (**) — правое.

Типы алгоритмов

- Нисходящий парсер (англ. *top-down parser*) — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.
    - LL-анализатор.
- Восходящий парсер (англ. *bottom-up parser*) — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом.
    - LR-анализатор.

LL-анализатор (LL parser) — нисходящий синтаксический анализатор для подмножества контекстно-свободных грамматик. Он анализирует входной поток слева

направо, и строит левый вывод грамматики (используются только левые порождения). Класс грамматик, для которых можно построить LL-анализатор, известен как LL-грамматики.

LL-анализатор называется LL(k)-анализатором, если данный анализатор использует предпросмотр на k токенов (лексем) при разборе входного потока. Грамматика, которая может быть распознана LL(k)-анализатором без возвратов к предыдущим символам, называется LL(k)-грамматикой. Язык, который может быть представлен в виде LL(k)-грамматики, называется LL(k)-языком.

LL(1)-грамматики очень распространены, потому что соответствующие им LL-анализаторы просматривают поток только на один символ вперед при принятии решения о том, какое правило грамматики необходимо применить. Языки, основанные на грамматиках с большим значением k, традиционно считались трудными для распознавания, хотя при широком распространении генераторов синтаксических анализаторов, поддерживающих LL(k) грамматики с произвольным k, это замечание уже неактуально.

LR-анализатор — синтаксический анализатор для исходных кодов программ, который читает входной поток слева (Left) направо и производит наиболее правую (Right) продукцию контекстно-свободной грамматики (правые порождения). Используется также термин LR(*k*)-анализатор, где *k* выражает количество непрочитанных символов предпросмотра во входном потоке, на основании которых принимаются решения при анализе. Обычно *k* равно 1 и часто опускается.

Говорится, что LR-анализатор выполняет разбор снизу вверх, потому что он пытается вывести продукцию верхнего уровня грамматики, строя её из *листов*.

LR-анализ может применяться к большему количеству языков, чем LL-анализ, а также лучше в части сообщения об ошибках, то есть он определяет синтаксические ошибки там, где вход не соответствует грамматике, как можно раньше. В отличие от этого, LL(k) анализаторы могут задерживать определение ошибки до другой ветки грамматики из-за отката, часто затрудняя определение места ошибки в местах общих длинных префиксов.

LR-анализаторы сложно создавать вручную и обычно они создаются генератором синтаксических анализаторов или компилятором компиляторов.

Левая рекурсия

Так вот, возвращаясь к примеру, рассмотренному выше:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

Эта грамматика относится к классу LR-грамматик и может быть разобрана восходящим анализатором. Для нисходящего анализа она не подходит, поскольку в ней имеется левая рекурсия. Грамматика является леворекурсивной (left recursive), если в ней имеется нетерминал A, такой, что существует порождение A -> Aa для некоторой строки a.

Методы нисходящего разбора не в состоянии работать с леворекурсивными грамматиками, поэтому требуется преобразование грамматики, которое устранило бы из нее левую рекурсию.

Леворекурсивная пара продукций A -> Aa | b может быть заменена следующими нелеворекурсивными продукциями:

A -> b A’

A’ -> aA’ | ε

без изменения строк, порождаемых из A. Одного этого правила достаточно для большого количества грамматик.

Пример

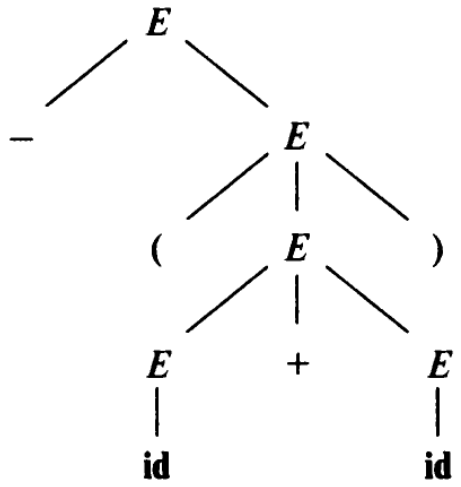
Рассмотрим, как можно устранить левую рекурсию из описанной выше грамматики с помощью данного правила. Для начала нужно заменить пару продукций E -> E + T | T следующими двумя продукциями: E -> T E’ и E’ -> +T E’ | εыва. Новые продукции для T и T’ получаются аналогично при устранении левой рекурсии из продукции T -> T * F | F. В итоге получаем следующие правила:

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *F\ T' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Деревья разбора и порождения

Дерево разбора может рассматриваться как графическое представление порождения, из которого удалена информация о порядке замещения нетерминалов. Каждый внутренний узел дерева разбора представляет применение продукции. Внутренний узел дерева помечен нетерминалом A из заголовка соответствующей продукции, а дочерние узлы слева направо — символами из тела продукции,использованной в порождении для замены A.

Например, дерево разбора для “-(id + id)”, показанное на рисунке ниже, получается как в результате порождения (*), так и в результате порождения (**), рассмотренных выше.



Листья дерева разбора помечены нетерминалами или терминалами и, будучи прочитаны слева направо, образуют так называемую крону (yield) или границу (frontier) дерева.

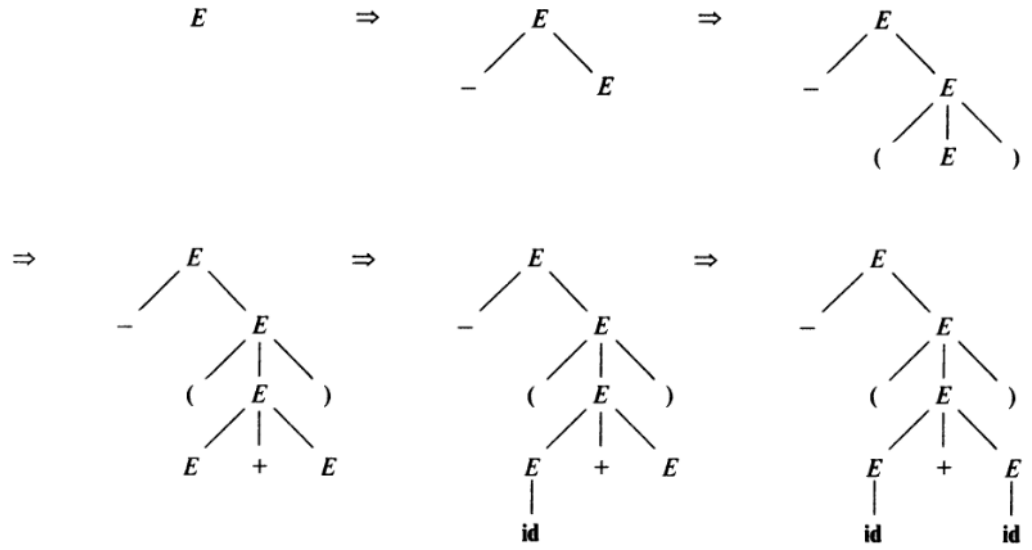
Пример

Последовательность деревьев разбора, построенная на основе порождения (*), показана на рисунке ниже.

Первый шаг этого порождения — правило “E => -E”. Для моделирования этого шага мы добавляем к корню E начального дерева два дочерних узла, помеченных “-” и “E”.

Вторым шагом порождения является “-E => -(E)”. Соответственно, мы добавляем три дочерних узла, помеченных (, E и), к листу E во втором дереве для получения третьего дерева с кроной -(E). Продолжая построения описанным способом, мы получим в качестве полного дерева разбора шестое дерево последовательности.

Поскольку дерево разбора игнорирует порядок, в котором производилось замещение символов в сентенциальной форме, между порождениями и деревьями разбора возникает соотношение "многие к одному". Например, и порождение (*), и порождение (**) связаны с одним и тем же окончательным деревом разбора, показанным на рисунке в самом конце.

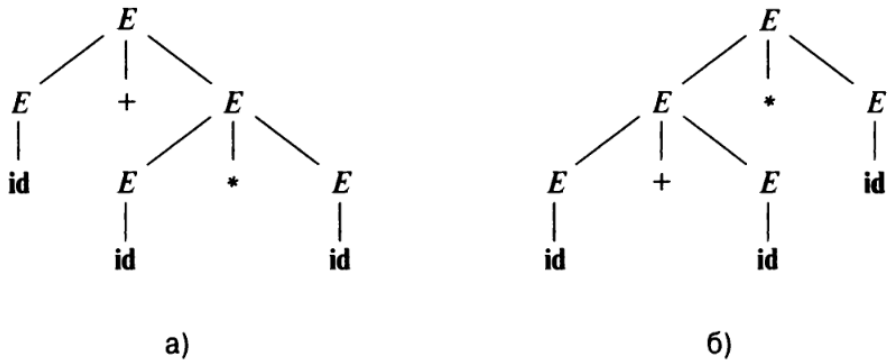


Неоднозначность

Грамматика для арифметических выражений допускает два разных левых порождения для предложения **id+id*id**:

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$

Соответствующие деревья разбора показаны на рисунке:



а) б)

Обратите внимание, что дерево на рисунке слева отражает обычно принимаемые приоритеты операторов + и *, в то время как дерево на рисунке справа отражает обратное

соотношение приоритетов. Иными словами, обычно считается, что приоритет оператора * выше, чем приоритет оператора +, так что обычно выражение наподобие a+b*c вычисляется как a+(b*c), а не как (a+b)*c.

Для большинства синтаксических анализаторов грамматика должна быть однозначной, поскольку, если это не так, мы не в состоянии определить дерево разбора для предложения единственным образом.

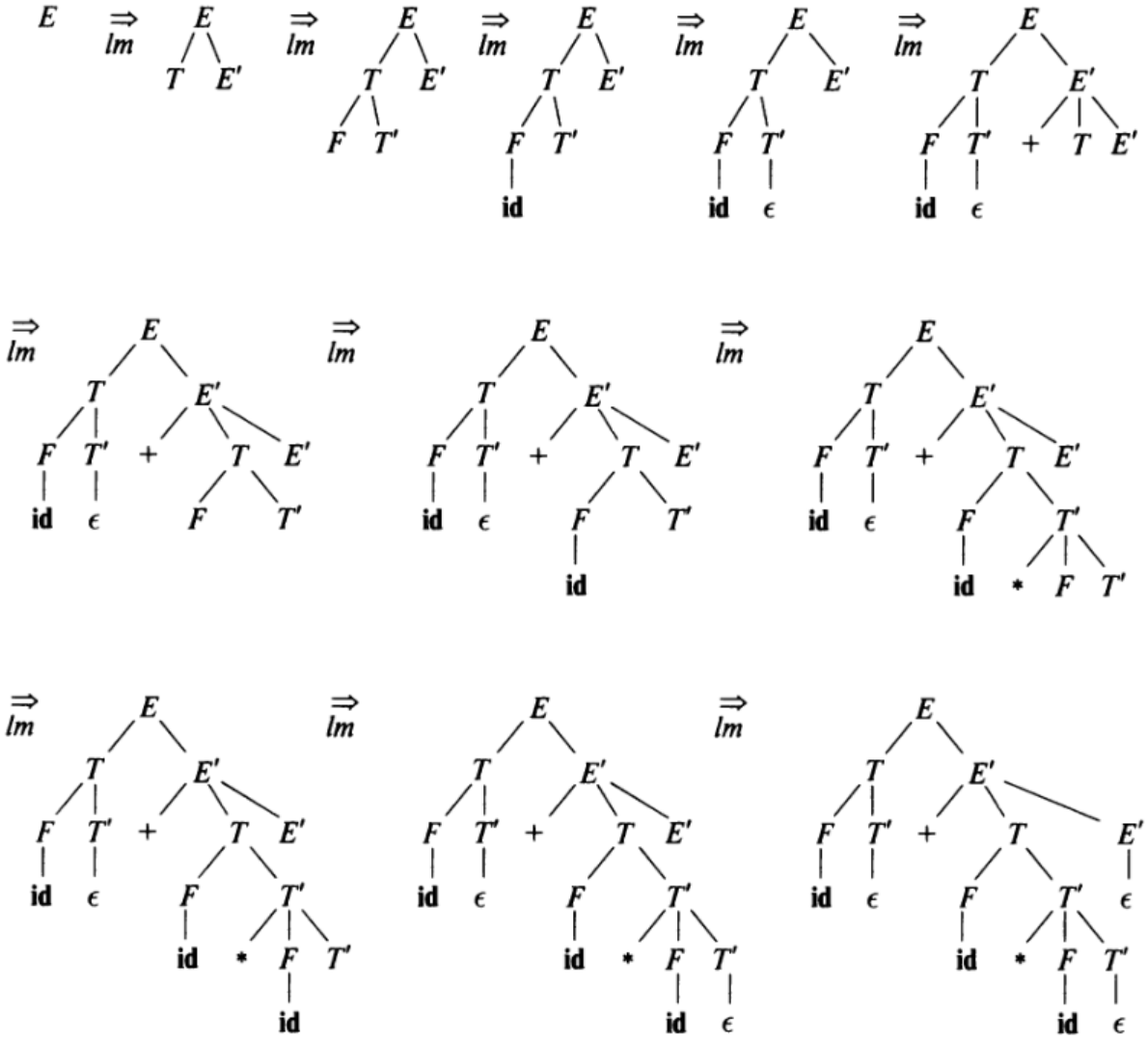
Нисходящий синтаксический анализ

Нисходящий синтаксический анализ можно рассматривать как задачу построения дерева разбора для входной строки, начиная с корня и создавая узлы дерева разбора в прямом порядке обхода. Или, что то же самое, нисходящий синтаксический анализ можно рассматривать как поиск левого порождения входной строки.

Пример

На рисунке ниже приведена последовательность деревьев разбора для входной строки “id+id*id”, представляющая собой нисходящий синтаксический анализ в соответствии с уже разбиравшейся сегодня грамматикой:

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *F\ T' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$



Эта последовательность деревьев соответствует левому порождению входной строки.

На каждом шаге нисходящего синтаксического анализа ключевой проблемой является определение продукции, применимой для нетерминала, скажем, A. Когда A-продукция выбрана, остальная часть процесса синтаксического анализа состоит из проверки "соответствий" терминальных символов в теле продукции входной строке. Мы рассмотрим общий вид нисходящего разбора, называющийся синтаксическим анализом методом рекурсивного спуска и который может потребовать возврата (отката — backtracking) для поиска корректной A-продукции, которая должна быть применена.

Есть еще предиктивный синтаксический анализ — частный случай синтаксического анализа методом рекурсивного спуска, не требующего возврата. Он выбирает корректную A-продукцию путем предпросмотра фиксированного количества символов входной строки; типичной является ситуация, когда достаточно просмотреть только один (очередной)

входной символ.

Синтаксический анализ методом рекурсивного спуска

Программа синтаксического анализа методом рекурсивного спуска (recursive-descent parsing) состоит из набора процедур, по одной для каждого нетерминала. Работа программы начинается с вызова процедуры для стартового символаи успешно заканчивается в случае сканирования всей входной строки. Псевдокод для типичного нетерминала показан на рисунке ниже. Обратите внимание на то, что этот псевдокод недетерминированный, поскольку он начинается с выбора A-продукции для применения не указанным способом.

```
     void A() {
1)       Выбираем A-продукцию A → X1X2 ... Xk;
2)       for ( i от 1 до k ) {
3)           if ( Xi — нетерминал )
4)               Вызов процедуры Xi ();
5)           else if ( Xi равно текущему входному символу a )
6)               Переходим к следующему входному символу;
7)           else /* Обнаружена ошибка */;
         }
     }
```

Как говорилось ранее, рекурсивный спуск в общем случае может потребовать выполнения возврата, т.е. повторения сканирования входного потока. Чтобы разрешить возврат, псевдокод выше должен быть немного модифицирован. Во-первых, невозможно выбрать единственную A-продукцию в строке 1, так что требуется испытывать каждую из нескольких продукций в некотором порядке. Во-вторых, ошибка в строке 7 не является окончательной и предполагает возврат к строке 1 и испытание другой A-продукции. Объявлять о найденной во входной строке ошибке можно только в том случае, если больше не имеется непроверенных A-продукций. Чтобы быть в состоянии проверить новую A-продукцию, нужно иметь возможность сбросить указатель входного потока в состояние, в котором он находился при первом достижении строки 1. Таким образом, для хранения этого указателя входного потока требуется локальная переменная.

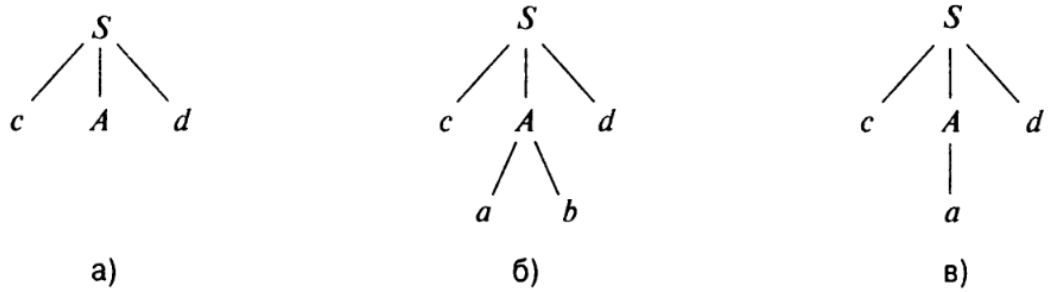
Пример

Рассмотрим следующую грамматику:

$$S \rightarrow c\,A\,d$$
$$A \rightarrow a\,b \mid a$$

Чтобы построить дерево разбора для входной строки w = cad, начнем с дерева, состоящего из единственного узла с меткой S, и указателя входного потока,

указывающего на c, первый символ w. S имеет единственную продукцию, так что мы используем ее для разворачивания S и получения показанного на рисунке а) дерева. Крайний слева лист, помеченный c, соответствует первому символу входного потока w, так что мы перемещаем указатель входного потока к a, второму символу w, и рассматриваем следующий лист, помеченный A.



а) б) в)

Теперь мы разворачиваем A с использованием первой альтернативы, A -> a b, и получаем таким образом дерево, показанное на рисунке б). У нас имеется совпадение второго входного символа, a, так что мы переходим к третьему символу, d, и сравниваем его с очередным листом b. Поскольку b не соответствует d, мы сообщаем об ошибке и возвращаемся к A, чтобы выяснить, нет ли альтернативной продукции, которая не была проверена до этого момента. Вернувшись к A, мы должны сбросить указатель входного потока так, чтобы он указывал на позицию 2, в которой мы находились, когда впервые столкнулись с A. Это означает, что процедура для A должна хранить указатель на входной поток в локальной переменной.

Вторая альтернатива для A дает дерево разбора, показанное на рисунке в). Лист a соответствует второму символу w, а лист d — третьему символу. Поскольку нами построено дерево разбора для входной строки w, мы завершаем работу и сообщаем об успешном выполнении синтаксического анализа и построении дерева разбора.

При реализации таких алгоритмов нужно всегда помнить, что леворекурсивная грамматика может привести синтаксический анализатор, работающий методом рекурсивного спуска, к бесконечному циклу, т.е. когда мы попытаемся развернуть нетерминал A, то в конечном счете можем найти этот же нетерминал и прийти к попытке развернуть A, так и не взяв ни одного символа из входного потока.