Курс «Компиляторные технологии»

Лекция 10

Синтаксический анализ (1)

Курносов Михаил Георгиевич

www.mkurnosov.net

Сибирский государственный университет телекоммуникаций и информатики
Весенний семестр

Разбор (parsing)

- Для любой контекстно-свободной грамматики существует анализатор, который требует для разбора строки из n терминалов время, не превышающее $O(n^3)$
- Для разбора почти всех встречающихся на практике языков программирования можно построить алгоритм с линейным временем разбора $O(n)$
- **Основные типы синтаксических анализаторов**:
  - **универсальные**: алгоритм Кока-Янгера-Касами (Cocke-Younger-Kasami), алгоритм Эрли (Earley), редко используются на практике из-за низкой эффективности
  - **восходящие** (bottom-up)
  - **нисходящие** (top-down)

Методы разбора
(по порядку построения узлов дерева разбора)

Нисходящие
(сверху вниз, top-down)

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

Восходящие
(снизу вверх, bottom-up)

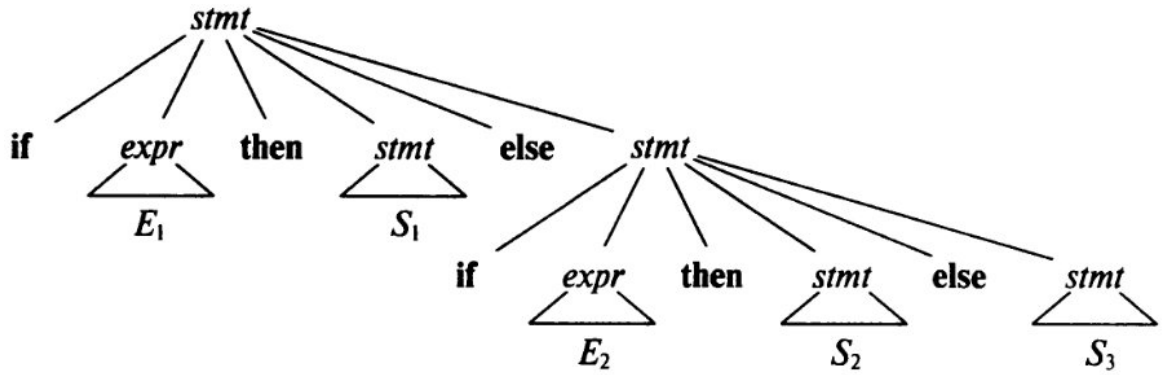
- Построение узлов дерева разбора от листьев к корню
- Применимы для большего класса грамматик
- Применяются в генераторах синтаксических анализаторов

Устранение неоднозначности грамматики

- Устраним неоднозначность из следующей грамматики с "висящим else"

$$
\begin{aligned}
stmt \;\rightarrow\; & \textbf{if } expr \textbf{ then } stmt \\
\mid\; & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
\mid\; & \textbf{other}
\end{aligned}
$$

- **Строка**: `if E1 then S1 else if E2 then S2 else S3`
- Дерево разбора:

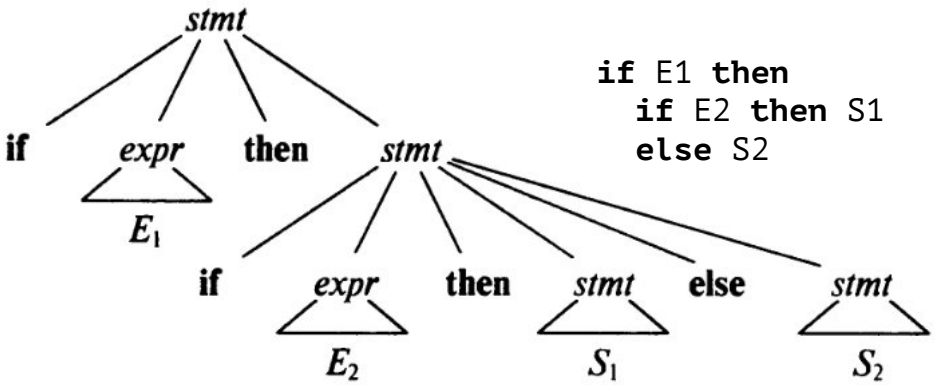


Устранение неоднозначности грамматики

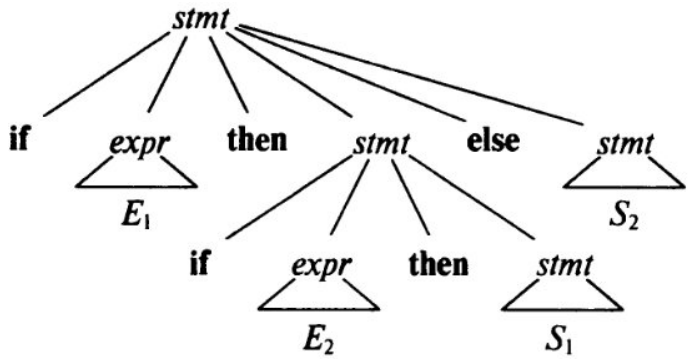
- Устраним неоднозначность из следующей грамматики с "висящим else"

$$
\begin{aligned}
stmt \;\rightarrow\; & \textbf{if } expr \textbf{ then } stmt \\
\mid\; & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
\mid\; & \textbf{other}
\end{aligned}
$$

- **Строка**: `if E1 then if E2 then S1 else S2`
- Два дерева разбора — неоднозначность:




```
if E1 then
  if E2 then S1
  else S2
```




```
if E1 then
  if E2 then S1
else
  S2
```

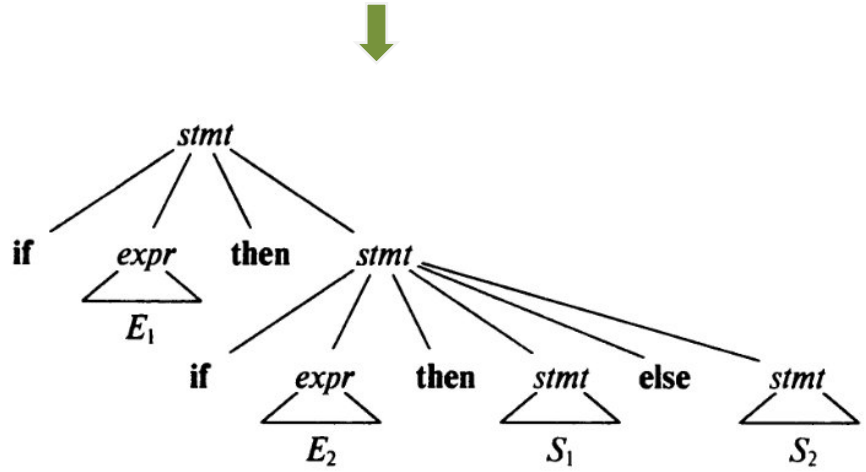
- В языках программирования с if-then-else такого вида предпочтительно первое дерево разбора
- Общее правило: «*сопоставить каждое else ближайшему незанятому then*»

Однозначная грамматика для инструкций if-then-else

- Ключевая идея — инструкция, появляющаяся между then и else, должна быть «сбалансированная» (matched) — не должна оканчиваться открытым или не соответствующим некоторому else ключевым словом then
- Сбалансированная инструкция может либо представлять собой полную инструкцию if-then-else, не содержащую открытых инструкций, либо быть любой инструкцией, отличающейся от условной

stmt	→	*matched_stmt*
	|	*open_stmt*
matched_stmt	→	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	**other**
open_stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *open_stmt*

```
if E1 then if E2 then S1 else S2
```



Устранение левой рекурсии

- Грамматика является *леворекурсивной* (left recursive), если в ней имеется нетерминал *A*, такой, что существует порождение *A* => *Aa* для некоторой строки *a*
- Методы нисходящего разбора (top-down) не могут работать с леворекурсивными грамматиками, требуется устранение левой рекурсии — переход к новой грамматике

- Непосредственная левая рекурсия (immediate left recursion)

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

заменена нелеворекурсивными продукциями

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Косвенная 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 (top-down) не могут работать с леворекурсивными грамматиками, требуется устранение левой рекурсии — переход к новой грамматике

- Косвенная левая рекурсия
- Как устранить?

$$
\begin{aligned}
S &\rightarrow A\,a \mid b \\
A &\rightarrow A\,c \mid S\,d \mid \epsilon
\end{aligned}
$$

Устранение левой рекурсии

Алгоритм 4.8. Устранение левой рекурсии

ВХОД: грамматика G без циклов и ϵ-продукций.

ВЫХОД: эквивалентная грамматика без левой рекурсии.

МЕТОД: применить алгоритм, приведенный на рис. 4.11. Обратите внимание, что получающаяся грамматика без левых рекурсий может иметь ϵ-продукции. □

1) Расположить нетерминалы в некотором порядке $A_1, A_2, \ldots, A_n$.
2) **for** (каждое i от 1 до n) {
3) **for** (каждое j от 1 до $i-1$) {
4) заменить каждую продукцию вида $A_i \rightarrow A_j$ продукциями $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, где $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ — все текущие A_j-продукции
5) }
6) устранить непосредственную левую рекурсию среди A_i-продукций
7) }

- Систематически удаляет из грамматики левую рекурсию
- Работает с грамматиками, не имеющими циклов: порождений типа A => ... => A и ε-продукций (A —> ε)

Устранение левой рекурсии

Алгоритм 4.8. Устранение левой рекурсии

ВХОД: грамматика G без циклов и ϵ-продукций.

ВЫХОД: эквивалентная грамматика без левой рекурсии.

МЕТОД: применить алгоритм, приведенный на рис. 4.11. Обратите внимание, что получающаяся грамматика без левых рекурсий может иметь ϵ-продукции. □

1) Расположить нетерминалы в некотором порядке $A_1, A_2, \ldots, A_n$.
2) **for** (каждое i от 1 до n) {
3) **for** (каждое j от 1 до $i-1$) {
4) заменить каждую продукцию вида $A_i \to A_j$ продукциями $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, где $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ — все текущие A_j-продукции
5) }
6) устранить непосредственную левую рекурсию среди A_i-продукций
7) }

(6) Заменена нелеворекурсивными продукциями

$$A \to A\alpha \mid \beta \longrightarrow \begin{aligned} A &\to \beta A' \\ A' &\to \alpha A' \mid \epsilon \end{aligned}$$

Пример

$$\begin{aligned} S &\to A\,a \mid b \\ A &\to A\,c \mid S\,d \mid \epsilon \end{aligned}$$

- Располагаем нетерминалы в порядке *S*, *A*
- *i* = 1: левой рекурсии среди *S* продукций нет
- *i* = 2: подставляем *S*-продукцию в *A* –> *S d*

$$A \to A\,c \mid A\,a\,d \mid b\,d \mid \epsilon$$

- Устраняем непосредственную левую рекурсию среди A-продукций

$$\begin{aligned} S &\to A\,a \mid b \\ A &\to b\,d\,A' \mid A' \\ A' &\to c\,A' \mid a\,d\,A' \mid \epsilon \end{aligned}$$

Левая факторизация (left factoring)

- В построенной грамматике может быть не ясно, какая из двух альтернативных продукций должна использоваться для нетерминала
- **Пример**: встретив во входном потоке if, мы не можем выбрать ни одну из продукций, нужны следующие символы потока

$$
\begin{aligned}
\textit{stmt} \quad &\rightarrow \quad \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
&\mid \quad \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}
\end{aligned}
$$

- **Левая факторизация** (left factoring) — преобразование грамматики в пригодную для предиктивного, или нисходящего, синтаксического анализа

$$
A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \longrightarrow \quad
\begin{aligned}
A &\rightarrow \alpha A' \\
A' &\rightarrow \beta_1 \mid \beta_2
\end{aligned}
$$

Левая факторизация (left factoring)

- **Алгоритм 4.10. Левая факторизация грамматики**
- Вход: грамматика G.
- Выход: эквивалентная левофакторизованная грамматика.

1. Для каждого нетерминала *A* находим самый длинный префикс α, общий для двух или большего числа альтернатив
2. Если α != ε, имеется нетривиальный общий префикс, заменим все продукции

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma,$$

   где γ представляет все альтернативы, не начинающиеся с α, продукциями

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_1 \mid \ldots \mid \beta_n$$

3. Выполняем это преобразование до тех пор, пока никакие две альтернативы нетерминала не будут иметь общий префикс

Левая факторизация (left factoring)

- **Алгоритм 4.10. Левая факторизация грамматики**
- Вход: грамматика G.
- Выход: эквивалентная левофакторизованная грамматика.

1. Для каждого нетерминала *A* находим самый длинный префикс α, общий для двух или большего числа альтернатив
2. Если α != ε, имеется нетривиальный общий префикс, заменим все продукции

$$A \text{ —> } \alpha\beta_1 \mid \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma,$$

где γ представляет все альтернативы, не начинающиеся с α, продукциями

$$A \text{ —> } \alpha A' \mid \gamma$$

$$A' \text{ —> } \beta_1 \mid \beta_1 \mid \ldots \mid \beta_n$$

3. Выполняем это преобразование до тех пор, пока никакие две альтернативы нетерминала не будут иметь общий префикс

```
if E then
  S
else
  S
```

$$S \rightarrow i\,E\,t\,S \mid i\,E\,t\,S\,e\,S \mid a$$
$$E \rightarrow b$$

$$S \rightarrow i\,E\,t\,S\,S' \mid a$$
$$S' \rightarrow e\,S \mid \epsilon$$
$$E \rightarrow b$$

Не контекстно-свободные языковые конструкции

- **Пример 1. Проверка того, что идентификаторы объявлены до их использования в программе**
- Язык L_1 состоит из строк вида *wcw*, где первое *w* представляет определение переменной *w*, а второе *w* — её использование

$$L_1 = \{wcw \mid w \in (\mathbf{a} \mid \mathbf{b})^*\}$$

- L_1 — не является контекстно-свободным языком, поэтому такие проверки в C/C++/Java/... выполняются на этапе семантического анализа

- **Пример 2. Проверка соответствия количества фактических параметров при вызове функции количеству формальных параметров в ее объявлении**
- Язык L_2 состоит из строк вида $a^n b^m c^n d^m$
- Здесь a^n и b^m могут представлять списки формальных параметров двух функций, объявленных с *n* и *m* аргументами, в то время как c^n и d^m — списки фактических параметров в вызовах этих двух функций

$$L_2 = \{a^n b^m c^n d^m \mid n \geqslant 1 \text{ и } m \geqslant 1\}$$

- L_2 — не является контекстно-свободным языком

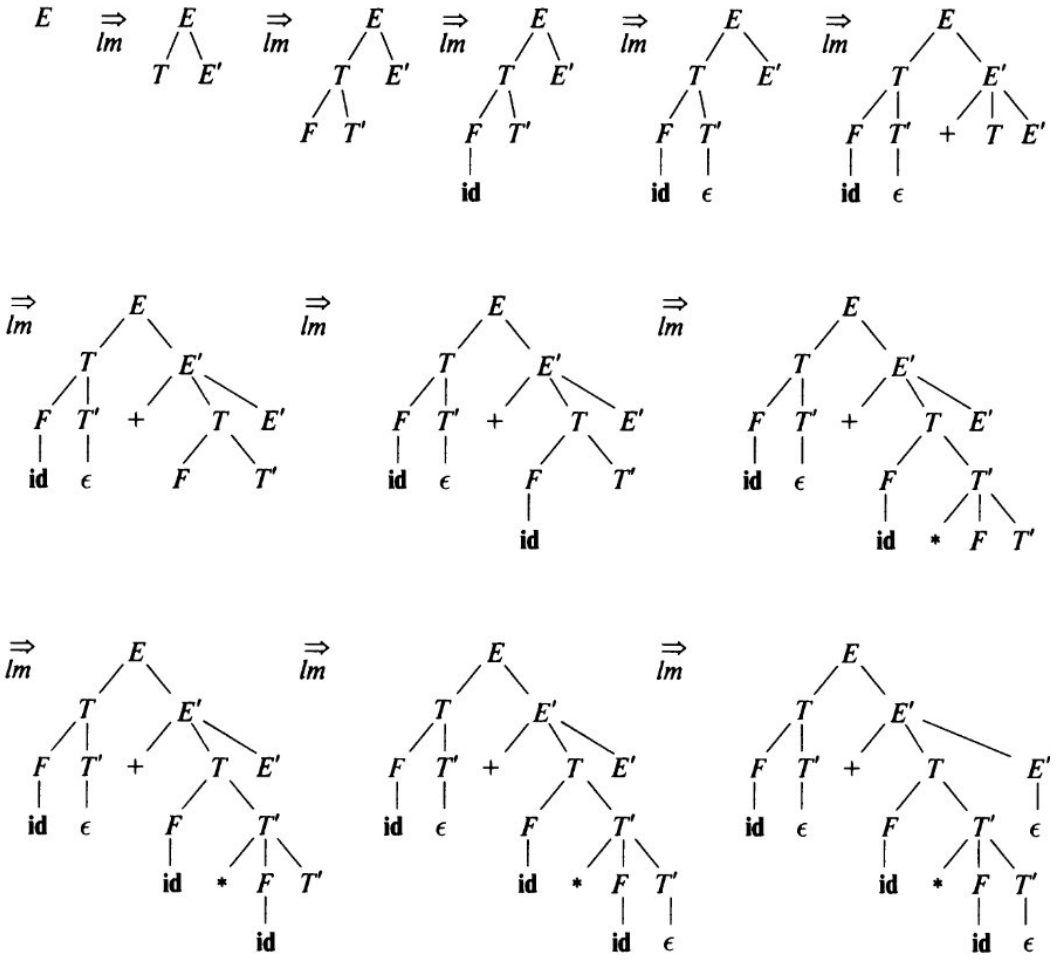
Нисходящий синтаксический анализ (top-down parsing)

- **Нисходящий синтаксический анализ** (top-down parsing) — построение дерева разбора для входной строки, начиная с корня и создавая узлы дерева разбора в прямом порядке обхода (обход в глубину: корень, левый потомок, правый потомок, pre-order traversal)

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

`id + id * id` ⟶

Последовательность деревьев разбора



Синтаксический анализ методом рекурсивного спуска

- Программа синтаксического анализа методом рекурсивного спуска (recursive-descent parsing) состоит из набора процедур, по одной для каждого нетерминала
- Работа начинается с вызова процедуры для стартового символа грамматики и успешно заканчивается в случае сканирования всей входной строки

- Псевдокод для типичного нетерминала

```
    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 */;
        }
    }
```

- псевдокод недетерминированный, начинается с выбора А-продукции для применения не указанным способом

Синтаксический анализ методом рекурсивного спуска

- Программа синтаксического анализа методом рекурсивного спуска (recursive-descent parsing) состоит из набора процедур, по одной для каждого нетерминала
- Работа начинается с вызова процедуры для стартового символа грамматики и успешно заканчивается в случае сканирования всей входной строки

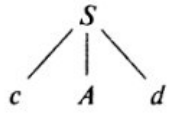
$$S \rightarrow c\,A\,d$$
$$A \rightarrow a\,b \mid a$$

- Псевдокод для типичного нетерминала

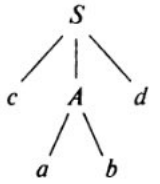
```
    void A() {
1)      Выбираем A-продукцию A → X1X2 ... Xk;
2)      for ( i от 1 до k ) {
3)          if ( Xi — нетерминал )
4)              Вызов процедуры Xi ();
5)          else if ( Xi равно текущему входному символу a )
6)              Переходим к следующему входному символу;
7)          else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

Разбор строки: *cad*

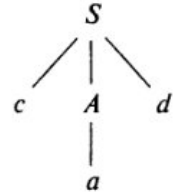
1. Крайний слева лист, помеченный *c*, соответствует первому символу входного потока, перемещаем указатель *a* и рассматриваем следующий лист *A*



2. Совпадение второго входного символа, *a*, переходим к третьему символу *d*
3. *b* не соответствует *d*, сообщаем об ошибке и возвращаемся (откат — backtracking) к *A*, чтобы выяснить, нет ли альтернативной продукции, которая не была проверена



4. Вернувшись к *A*, сбрасываем указатель на позицию 2, в которой мы находились, когда столкнулись с *A*
5. Вторая альтернатива для *A* — лист *a* соответствует второму символу потока, а лист *d* — третьему символу



- Леворекурсивная грамматика может привести синтаксический анализатор, работающий методом рекурсивного спуска, к бесконечному циклу — при разворачивании нетерминал *A*, то в конечном счете можем найти этот же нетерминал и прийти к попытке развернуть A

Функция FIRST

- В процессе нисходящего синтаксического анализа FIRST и FOLLOW позволяют выбрать применяемую продукцию на основании очередного символа входного потока
- FIRST(α) — множество терминалов, с которых начинаются строки, порождаемые α, где α — произвольная строка символов грамматики

- **Вычисление FIRST(α)**

1. Если X — терминал, то $\mathbf{FIRST}(X) = \{X\}$.

2. Если X — нетерминал и имеется продукция $X \to Y_1 Y_2 \ldots Y_k$ для некоторого $k \geqslant 1$, то поместим a в $\mathbf{FIRST}(X)$, если для некоторого i $a \in \mathbf{FIRST}(Y_i)$ и ϵ входит во все множества $\mathbf{FIRST}(Y_1)$, ..., $\mathbf{FIRST}(Y_{i-1})$, т.е. $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. Если ϵ входит в $\mathbf{FIRST}(Y_j)$ для всех $j = 1, 2, \ldots, k$, то добавляем ϵ к $\mathbf{FIRST}(X)$. Например, все, что находится в множестве $\mathbf{FIRST}(Y_1)$, есть и в множестве $\mathbf{FIRST}(X)$. Если Y_1 не порождает ϵ, то больше мы ничего не добавляем к $\mathbf{FIRST}(X)$, но если $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, то к $\mathbf{FIRST}(X)$ добавляется $\mathbf{FIRST}(Y_2)$ и т.д.

3. Если имеется продукция $X \to \epsilon$, добавим ϵ к $\mathbf{FIRST}(X)$.

Функция FOLLOW

- FOLLOW(*A*) — множество терминалов *a*, которые могут располагаться непосредственно справа от нетерминала A в некоторой сентенциальной форме
- FOLLOW(*A*) — множество терминалов *a*, таких, что существует порождение вида S => ... => α*Aa*β для некоторых α и β

- **Вычисление FOLLOW(*A*) для всех нетерминалов A**

1. Поместим \$ в FOLLOW (S), где S — стартовый символ, а \$ — правый ограничитель входного потока.
2. Если имеется продукция $A \to \alpha B \beta$, то все элементы множества FIRST (β), кроме ϵ, помещаются в множество FOLLOW (B).
3. Если имеется продукция $A \to \alpha B$ или $A \to \alpha B \beta$, где FIRST (β) содержит ϵ, то все элементы из множества FOLLOW (A) помещаются в множество FOLLOW (B).

Примеры для FIRST и FOLLOW

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

- FIRST(*F*) = FIRST(*T*) = FIRST(*E*) = { (, id }
- FIRST(*E'*) = { +, ε }
- FIRST(*T'*) = { *, ε }

- FOLLOW(*E*) = {), $ }
- FOLLOW(*T*) = FOLLOW(*T'*) = { +,), $ }
  В продукциях *T* всегда следует *E'*, следовательно все элементы множества FIRST(*E'*), кроме ε, должны находиться в FOLLOW(*T*)
- FOLLOW(*F*) = { +, *,), $ }

LL(1)-грамматики

- **Предиктивные синтаксические анализаторы** (на базе рекурсивного спуска без возврата) могут быть построены для класса **LL(1)-грамматик**
  - **L**L(1): первое L — сканирование входного потока слева направо
  - L**L**(1): второе L — получение левого порождения (leftmost derivation)
  - LL(**1**) — использование на каждом шаге предпросмотра (lookahead) одного символа для принятия решения о действиях синтаксического анализатора
- **В LL(1)-грамматике не может быть ни левой рекурсии, ни неоднозначности**
- **Определение**. Грамматика *G* принадлежит классу LL(1) тогда и только тогда, когда для любых двух различных продукций A —> α | β выполняются следующие условия:
  1. Не существует такого терминала *a*, для которого и α, и β порождают строку, начинающуюся с *a*
  2. Пустую строку может порождать не более чем одна из продукций α или β
  3. Если β => ... => ε, то α не порождает ни одну строку, начинающуюся с терминала из FOLLOW(A)
  4. Если α => ... => ε, то β не порождает ни одну строку, начинающуюся с терминала из FOLLOW(A)

FIRST(α) и FIRST(β) — непересекающиеся множества

LL(1)-грамматики

- **Для LL(1)-грамматики может быть построен предиктивный синтаксический анализатор** – корректная продукция для применения к нетерминалу может быть выбрана путем просмотра только текущего входного символа
- Языковые конструкции управления потоком (control flow) с их определяющими ключевыми словами обычно удовлетворяют ограничениям LL(1)

$$
\begin{aligned}
stmt \rightarrow\ & \textbf{if}\ (expr)\ stmt\ \textbf{else}\ stmt \\
\mid\ & \textbf{while}\ (expr)\ stmt \\
\mid\ & \{stmt\_list\}
\end{aligned}
$$

- Ключевые слова **if**, **while** и символ { однозначно определяют, какая из альтернатив должна быть выбрана

Пример не LL(1)-грамматики

- **В LL(1)-грамматике не может быть ни левой рекурсии, ни неоднозначности**

- **Определение**. Грамматика *G* принадлежит классу LL(1) тогда и только тогда, когда для любых двух различных продукций A —> α | β выполняются следующие условия:
  1. Не существует такого терминала *a*, для которого и α, и β порождают строку, начинающуюся с *a*
  2. Пустую строку может порождать не более чем одна из продукций α или β
  3. Если β => ... => ε, то α не порождает ни одну строку, начинающуюся с терминала из FOLLOW(A)
  4. Если α => ... => ε, то β не порождает ни одну строку, начинающуюся с терминала из FOLLOW(A)

```
if E then
  S
else
  S
```

$$
\begin{aligned}
S &\rightarrow i\,E\,t\,S\,S' \mid a \\
S' &\rightarrow e\,S \mid \epsilon \\
E &\rightarrow b
\end{aligned}
$$

- **Грамматика неоднозначна**
- Неоднозначность проявляется в выборе продукции при встрече в потоке *e* (**else**)

Диаграмма переходов на основе грамматики

- **Диаграммы переходов** (transition diagram) полезны для визуализации предиктивных синтаксических анализаторов
- **Построение диаграммы переходов на основе грамматики**:
  1. Удалить левую рекурсию
  2. Выполнить левую факторизацию грамматики
  3. Для каждого нетерминала A:
     A. Создать начальное и конечное состояния
     B. Для каждой продукции $A \rightarrow X_1X_2...X_k$ создать путь из начального в конечное состояние с дугами, помеченными X_1, X_2, ..., X_k (или ε, если $A \rightarrow \varepsilon$)

- По диаграмме для каждого нетерминала
- Метки ребер — терминалы (токены) или нетерминалами
- Переход для терминала — переход выполняется, если этот токен будет очередным входным символом
- Переход для нетерминала A — вызов процедуры для A

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

Диаграммы переходов для нетерминалов E и E'

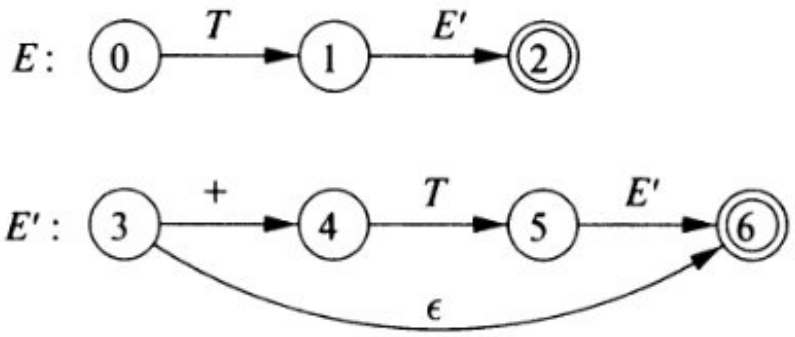


Таблица предиктивного синтаксического анализа (predictive parsing table)

- **Таблица предиктивного синтаксического анализа** *M*[*A*, *a*] — продукция, при помощи которой выполняется разворачивание нетерминала *A* (*A* — нетерминал, *a* — терминал или символ $)
- Алгоритм 4.17 может быть применен для получения таблицы *M* к любой грамматике *G*

Алгоритм 4.17. Построение таблицы предиктивного синтаксического анализа

ВХОД: грамматика G.

ВЫХОД: таблица синтаксического анализа M.

МЕТОД: для каждой продукции грамматики $A \to \alpha$ выполняем следующие действия.

1. Для каждого терминала a из $\text{FIRST}(\alpha)$ добавляем $A \to \alpha$ в ячейку $M[A, a]$.
2. Если $\epsilon \in \text{FIRST}(\alpha)$, то для каждого терминала b из $\text{FOLLOW}(A)$ добавляем $A \to \alpha$ в $M[A, b]$. Если $\epsilon \in \text{FIRST}(\alpha)$ и $\$ \in \text{FOLLOW}(A)$, то добавляем $A \to \alpha$ также и в $M[A, \$]$.

- Если после выполнения алгоритма ячейка *M*[*A*, *a*] осталась без продукции (пустая запись таблицы), устанавливаем ее значение равным **error**

Таблица предиктивного синтаксического анализа (predictive parsing table)

Алгоритм 4.17. Построение таблицы предиктивного синтаксического анализа

ВХОД: грамматика G.

ВЫХОД: таблица синтаксического анализа M.

МЕТОД: для каждой продукции грамматики $A \to \alpha$ выполняем следующие действия.

1. Для каждого терминала a из $\text{FIRST}(\alpha)$ добавляем $A \to \alpha$ в ячейку $M[A, a]$.
2. Если $\epsilon \in \text{FIRST}(\alpha)$, то для каждого терминала b из $\text{FOLLOW}(A)$ добавляем $A \to \alpha$ в $M[A, b]$. Если $\epsilon \in \text{FIRST}(\alpha)$ и $\$ \in \text{FOLLOW}(A)$, то добавляем $A \to \alpha$ также и в $M[A, \$]$.

$$E \to T\ E'$$

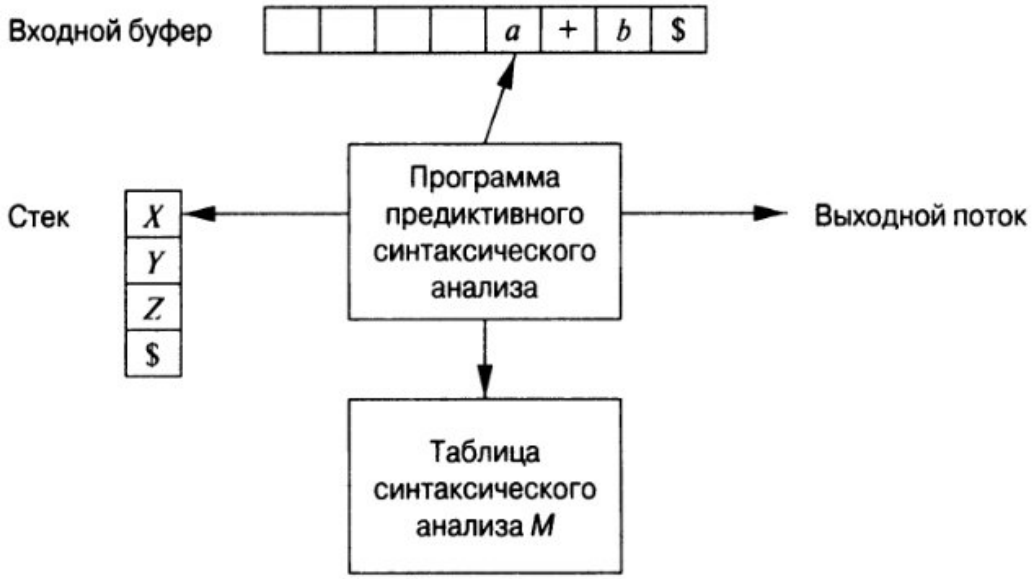
$$\text{FIRST}(T\ E') = \text{FIRST}(T) = \{(, \textbf{id}\}$$

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

НЕТЕР-МИНАЛ	ВХОДНОЙ СИМВОЛ					
	id	+	*	(	)	$
E	$E \to T\ E'$			$E \to T\ E'$		
E'		$E' \to +T\ E'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to F\ T'$			$T \to F\ T'$		
T'		$T' \to \epsilon$	$T' \to *\ F\ T'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \textbf{id}$			$F \to (E)$		

Нерекурсивный предиктивный синтаксический анализ

- Нерекурсивный предиктивный синтаксический анализатор: явное использования стека (структура данных) и таблицы синтаксического анализа
- Синтаксический анализатор имитирует левое порождение
- Синтаксический анализатор рассматривает символ на вершине стека *X* и текущий входной символ *a*
- Если *X* является нетерминалом, синтаксический анализатор выбирает *X*-продукцию в соответствии с записью *M*[*X*, *a*] таблицы синтаксического анализа (может выполняться дополнительный код — построения узла дерева разбора)
- Если *X* является терминаломВ — проверяется соответствие между терминалом *X* и текущим входным символом *a*
- Поведение синтаксического анализатора может быть описано в терминах его ***конфигураций*** (configuration), которые дают содержимое стека и оставшийся входной поток



Предиктивный синтаксический анализ, управляемый таблицей (table-driven predictive parsing)

ВХОД: строка w и таблица синтаксического анализа M для грамматики G.

ВЫХОД: если $w \in L(G)$ — левое порождение w; в противном случае — сообщение об ошибке.

```
Устанавливаем указатель входного потока ip так,
чтобы он указывал на первый символ строки w;
Устанавливаем X равным символу на вершине стека;
while ( X ≠ $ ) { /* Стек не пуст */
    Устанавливаем a равным символу, на который
        в настоящий момент указывает ip
    if ( X равен a ) {
        Снимаем символ со стека и перемещаем ip
            к следующему символу строки;
    }
    else if ( X — терминал ) error();
    else if ( M[X, a] — запись об ошибке ) error();
    else if ( M[X, a] = X → Y1Y2 ... Yk ) {
        Выводим продукцию X → Y1Y2 ... Yk;
        Снимаем символ со стека;
        Помещаем в стек Yk, Yk−1, ..., Y1; Y1
            помещается на вершину стека;
    }
    Устанавливаем X равным символу на вершине стека;
}
```

Начальное состояние

- входная строка: *w*$
- стек: стартовый символом *S* грамматики

Предиктивный синтаксический анализ, управляемый таблицей (table-driven predictive parsing)

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

Входная строка: → **id + id * id $**

Таблица предиктивного синтаксического анализа

Нетерминал	Входной символ					
	id	+	*	(	)	$
E	$E \rightarrow T\ E'$			$E \rightarrow T\ E'$		
E'		$E' \rightarrow +T\ E'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow F\ T'$			$T \rightarrow F\ T'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *\ F\ T'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Шаги алгоритма синтаксического анализа:

$$E \underset{lm}{\Rightarrow} T\ E' \underset{lm}{\Rightarrow} F\ T'\ E' \underset{lm}{\Rightarrow} \mathbf{id}\ T'\ E' \underset{lm}{\Rightarrow} \mathbf{id}\ E' \underset{lm}{\Rightarrow} \mathbf{id} + T\ E' \underset{lm}{\Rightarrow} \cdots$$

Соответствие	Стек	Входная строка	Действие
	$E\$$	**id** $+$ **id** $*$ **id**$	
	$T\ E'\$$	**id** $+$ **id** $*$ **id**$	Вывод $E \rightarrow T\ E'$
	$F\ T'\ E'\$$	**id** $+$ **id** $*$ **id**$	Вывод $T \rightarrow F\ T'$
	id $T'\ E'\$$	**id** $+$ **id** $*$ **id**$	Вывод $F \rightarrow$ **id**
id	$T'\ E'\$$	$+$**id** $*$ **id**$	Соответствие **id**
id	$E'\$$	$+$**id** $*$ **id**$	Вывод $T' \rightarrow \epsilon$
id	$+\ T\ E'\$$	$+$**id** $*$ **id**$	Вывод $E' \rightarrow +\ T\ E'$
id$+$	$T\ E'\$$	**id** $*$ **id**$	Соответствие $+$
id$+$	$F\ T'\ E'\$$	**id** $*$ **id**$	Вывод $T \rightarrow F\ T'$
id$+$	**id** $T'\ E'\$$	**id** $*$ **id**$	Вывод $F \rightarrow$ **id**
id $+$ **id**	$T'\ E'\$$	$*$**id**$	Соответствие **id**
id $+$ **id**	$*\ F\ T'\ E'\$$	$*$**id**$	Вывод $T' \rightarrow *\ F\ T'$
id $+$ **id**$*$	$F\ T'\ E'\$$	**id**$	Соответствие $*$
id $+$ **id**$*$	**id** $T'\ E'\$$	**id**$	Вывод $F \rightarrow$ **id**
id $+$ **id** $*$ **id**	$T'\ E'\$$	$	Соответствие **id**
id $+$ **id** $*$ **id**	$E'\$$	$	Вывод $T' \rightarrow \epsilon$
id $+$ **id** $*$ **id**	$	$	Вывод $E' \rightarrow \epsilon$

Восстановление после ошибок в предиктивном синтаксическом анализе

Вход: строка w и таблица синтаксического анализа M для грамматики G.

Выход: если $w \in L(G)$ — левое порождение w; в противном случае — сообщение об ошибке.

Устанавливаем указатель входного потока *ip* так,
чтобы он указывал на первый символ строки w;
Устанавливаем X равным символу на вершине стека;
while ($X \neq \$$) { /* Стек не пуст */
    Устанавливаем a равным символу, на который
        в настоящий момент указывает *ip*
    if (X равен a) {
        Снимаем символ со стека и перемещаем *ip*
            к следующему символу строки;
    }
    else if (X — терминал) *error*();
    else if ($M[X, a]$ — запись об ошибке) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$) {
        Выводим продукцию $X \rightarrow Y_1 Y_2 \ldots Y_k$;
        Снимаем символ со стека;
        Помещаем в стек $Y_k, Y_{k-1}, \ldots, Y_1$; Y_1
            помещается на вершину стека;
    }
    Устанавливаем X равным символу на вершине стека;
}

- Восстановление после ошибки **«в режиме паники»** — пропуск символов из входного потока до тех пор, пока не будет обнаружен токен из предопределенного множества синхронизирующих токенов

- **Варианты**

1. В синхронизирующее множество для нетерминала *A* помещаются все символы из множества FOLLOW(*A*)
2. В синхронизирующее множество помещаются все символы из множества FOLLOW(*A*) + ключевые слова, с которых начинаются инструкции, в синхронизирующие множества нетерминалов
3. Если добавить символы из FIRST(*A*) в синхронизирующее множество для нетерминала *A*, станет возможным продолжение анализа в соответствии с *A*, когда во входном потоке появится символ из FIRST(*A*)

Восстановление после ошибок в предиктивном синтаксическом анализе

Нетер-минал	Входной символ					
	id	+	*	(	)	$
E	$E \to T\ E'$			$E \to T\ E'$	*synch*	*synch*
E'		$E' \to +\ T\ E'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to F\ T'$	*synch*		$T \to F\ T'$	*synch*	*synch*
T'		$T' \to \epsilon$	$T' \to *\ F\ T'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	*synch*	*synch*	$F \to (E)$	*synch*	*synch*

- Если анализатор встречает пустую ячейку *M*[*A*, *a*], то входной символ *a* пропускается
- Если в этой ячейке находится запись *synch*, то нетерминал снимается с вершины стека в попытке продолжить синтаксический анализ
- Если токен на вершине стека не соответствует входному символу, то снимаем его со стека

Восстановление после ошибок в предиктивном синтаксическом анализе

Вход с ошибкой:
)id * + id $

Нетер-минал	Входной символ					
	id	+	*	(	)	$
E	$E \to T\ E'$			$E \to T\ E'$	*synch*	*synch*
E'		$E' \to +\ T\ E'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to F\ T'$	*synch*		$T \to F\ T'$	*synch*	*synch*
T'		$T' \to \epsilon$	$T' \to *\ F\ T'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	*synch*	*synch*	$F \to (E)$	*synch*	*synch*

Стек	Входная строка	Примечание
E\$	)**id** $*$ +**id**\$	Ошибка, пропускаем)
E\$	**id** $*$ +**id**\$	**id** $\in$ FIRST (E)
$T\ E'$\$	**id** $*$ +**id**\$	
$F\ T'\ E'$\$	**id** $*$ +**id**\$	
id $T'\ E'$\$	**id** $*$ +**id**\$	
$T'\ E'$\$	$*$ + **id**\$	
$*\ F\ T'\ E'$\$	$*$ + **id**\$	
$F\ T'\ E'$\$	+**id**\$	Ошибка, $M\ [F, +] = synch$
$T'\ E'$\$	+**id**\$	F снимается со стека
E'\$	+**id**\$	
$+\ T\ E'$\$	+**id**\$	
$T\ E'$\$	**id**\$	
$F\ T'\ E'$\$	**id**\$	
id $T'\ E'$\$	**id**\$	
$T'\ E'$\$	\$	
E'\$	\$	
\$	\$	