4. Теория языков

- Различные способы задания языков в компиляции:
  - Грамматики
  - Конечные и магазинные автоматы
- Соотношения между различными способами задания языков
- Приложения в компиляции

Лекция 4. Теория языков

В этой лекции рассматриваются следующие темы:

- Различные способы задания языков в компиляции:
  - Грамматики
  - Конечные и магазинные автоматы
- Соотношения между различными способами задания языков
- Приложения этой теории в компиляции

Задача определения языка

- Как определить используемый язык?
- Определение должно быть удобным, т.е.:
  - Определение должно быть конечным
  - Должен существовать алгоритм, за конечное число шагов проверяющий принадлежность некоторой входной цепочки языку
- Наиболее распространенные формализмы для задания языков: грамматики, регулярные выражения, конечные и магазинные автоматы, машины Тьюринга

Задача определения языка

Одна из первых задач, возникающих в процессе компиляции – это определение рассматриваемого языка программирования. При рассмотрении языков, состоящих из конечного множества цепочек, проще всего явным образом перечислить все допустимые входные цепочки. Но что делать с языками, не вводящими никаких ограничений на длину входной цепочки? Для потенциально бесконечных языков нам потребуется ввести какой-то конструктивный способ описания, который позволит нам задать правила, описывающие порождаемый ими язык. Такое описание должно удовлетворять некоторым свойствам:

- Само описание должно иметь конечную длину
- Для данного описания языка должен существовать алгоритм, который мог бы проверить принадлежность некоторой входной цепочки языку

Существует целый ряд математических формализмов, в той или иной степени удобных для задания языков – вообще, этап анализа входной программы наиболее разработан и лучше всего поддержан математическими теориями. Наиболее распространенным механизмом являются грамматики, которые задают все подходящие цепочки языка с помощью некоторых порождающих правил. Очевидное достоинство грамматик заключается в том, что существует множество систем, которые по заданной грамматике генерируют программу, проверяющую соответствие входной цепочки определяемому языку. Более того, полезную работу синтаксического анализатора (например, построение дерева разбора) можно проводить параллельно с самим распознаванием языка.

Другая часто используемая идея заключается в том, что создается некоторый обобщенный алгоритм, проверяющий за конечное число шагов принадлежность данной цепочки языку. Такой алгоритм либо останавливается после конечного числа шагов и говорит "да", либо останавливается и говорит "нет". Теоретически, нас могло бы устроить и зацикливание алгоритма на неподходящих входных цепочках, но на практике такое поведение не совсем удобно.

Определение грамматики

Грамматика определяется путем задания следующих компонент:

- Терминальные символы (алфавит)
- Нетерминальные символы (иногда также называемые понятия)
- Правила вывода
- Начальный символ

Определение грамматики

Грамматики представляют собой наиболее распространенный класс описаний языков. При описании грамматики необходимо начать с определения алфавита языка, который задается как набор допустимых *терминальных* символов. Кроме того, необходимо определить набор *правил вывода* вида $\alpha \rightarrow \beta$, с помощью которых строятся все цепочки языка. В левой и правой части этих правил могут встречаться специальные *нетерминальные* символы; в процессе вывода нетерминальные символы заменяются с помощью соответствующих правил до полной замены на соотвествующие терминалы. Наконец, грамматика должна включать в себя *начальный символ*, или аксиому, с которой начинается получение любого предложения языка.

Для формального определения грамматики нам потребуются следующие обозначения. Если A есть алфавит, то A^* обозначает множество всех строк (включая пустую строку ε), составленных из символов, входящих в A. Аналогично, A^+ определяет множество всех строк, составленных из символов, входящих в A, но без пустой строки. Нетерминалы мы будем обозначать прописными буквами, а терминалы – строчными. Итак, грамматика G определяется как следующая четверка: $G = (V_T, V_N, P, S)$, где

- V_T – конечное множество терминальных символов;
- V_N – не пересекающееся с V_T конечное множество нетерминальных символов;
- P – конечный набор порождающих правил вида (α, β), где $\alpha \in V^+, \beta \in V^*$
- S – начальный символ, где $S \in V_N$

Например, грамматикой, порождающей язык $\{0^n1^n \mid n \geq 0\}$, является G_0: $G_0 = (\{0,1\}, \{S\}, P, S)$, где $P = \{S \rightarrow 0S1,\ S \rightarrow \varepsilon\}$. Другой пример: рассмотрим грамматику, порождающую язык $\{a^m b^n \mid m,n \geq 0\}$. Такая грамматика имеет вид $G_1 = (\{a,b\}, \{S,A,B\}, P, S)$, где набор правил определяется следующим образом: $P = \{S \rightarrow AB,\ A \rightarrow aA,\ A \rightarrow \varepsilon,\ B \rightarrow bB,\ B \rightarrow \varepsilon\}$.

Определим также понятие выводимости: если αβγ – цепочка, состоящая из символов языка G, а β→δ – правило языка G, то $\alpha\beta\gamma \Rightarrow_G \alpha\delta\gamma$ (αδγ *непосредственно выводима из* αβγ *в G*). Рефлексивное и транзитивное замыкание этого отношения обозначим как $\alpha \Rightarrow^*_G \beta$ (*цепочка β выводима из α*; имя грамматики можно опускать для краткости).

Некоторые свойства грамматик

- Различные грамматики могут порождать один и тот же язык. Такие грамматики называются *эквивалентными*
- Левые части правил могут содержать другие нетерминалы (помимо определяемого)
- По соглашению, можно объединять несколько правил с одинаковой левой частью, записывая правые части через символ '|' ("или")

Некоторые свойства грамматик

Отметим некоторые свойства грамматик, которые нам придется учитывать в дальнейшем при построении грамматик реальных языков программирования.

Во-первых, различные грамматики могут порождать один и тот же язык (такие грамматики называются *эквивалентными*). Например, приведенная выше грамматика G_1 эквивалентна следующей грамматике $G_2 = (\{a,b\},\{S,Y\},P,S)$, где правила из P определены следующим образом: P = {*S→aS, S→a, S→b, S→bY, Y→b, Y→bY, S→ε*}.

Несмотря на эквивалентность определяемых языков, одна грамматика может быть значительно удобнее другой с точки зрения ее использования в компиляторе. Поэтому в дальнейшем мы будем рассматривать некоторые приемы преобразования грамматик с целью улучшения удобства работы с языком.

Во-вторых, необходимо отметить, что определение грамматик не накладывает никаких ограничений на количество нетерминалов в левой части правил и приведенные выше примеры не должны создавать обманчивого впечатления, что все грамматики содержат один и только один нетерминал в левой части каждого правила. В качестве иллюстрации приведем следующий пример: $G_3 = (\{a,b,c\},\{S,B,C\},P,S)$, где P содержит следующие правила: P = {*S→aSBC, S→abC, CB→BC, bB→bb, bC→bc, cC→cc*}. Это абсолютно законная грамматика, порождающая язык $\{a^n b^n c^n, n \geq 1\}$.

В качестве другого примера приведем еще одну грамматику, эквивалентную грамматике G_1: $G_4 = (\{0,1\},\{A,S\},P,S)$, где P = {*S→0A1, 0A→00A1, S→ε*}. В этом варианте левая часть одного из правил содержит пару из терминального и нетерминального символа.

В-третьих, упомянем одно соглашение, которое нам пригодится впоследствии: для обозначения *n* правил вида $\alpha \to \beta_1, \alpha \to \beta_2, \ldots, \alpha \to \beta_n$ мы будем использовать следующую запись: $\alpha \to \beta_1 | \beta_2 | \ldots | \beta_n$. В правилах такого вида вертикальная черта читается как "или".

Иерархия Хомского

- Включающая иерархия языков:
  - Грамматика без ограничений
  - Контекстно-зависимая грамматика
  - Контекстно-свободная грамматика
  - Выровненные влево или вправо грамматики (они же регулярные, лево- или праволинейные)
- Чем меньше ограничений накладывается на грамматику, тем более сложный язык программирования можно задать с ее помощью

Иерархия Хомского

Итак, мы убедились, что существует множество различных видов грамматик и, предположительно, некоторые из них могут оказаться удобнее для наших целей. Поэтому мы введем классификацию грамматик согласно их внешнему виду (эта классификация известна как *иерархия Хомского*, по фамилии автора).

Грамматика G называется:

- *выровненной вправо (праволинейной)*, если любое правило из P имеет вид $A \to xB$ или $A \to x$, где A, B – нетерминалы, а x – терминал (возможно, пустой).
- *контекстно-свободной (бесконтекстной)*, если любое правило из P имеет вид $A \to \alpha$, где A – нетерминал, α – нетерминал или терминал
- *контекстно-зависимой (неукорачивающей)*, если все правила из P имеет вид $\alpha \to \beta$, где $|\alpha| \le |\beta|$
- *общего вида (без ограничений)*, если грамматика не удовлетворяет ни одному из указанных выше ограничений.

Иногда приведенные выше классы нумеруют от трех до нуля и называют каждый класс "грамматикой типа *n*", например, грамматика общего вида называется грамматикой типа 0. Мы будем избегать этого обозначения, так как оно не проясняет суть вопроса.

Очевидно, что эта классификация – включающая, т.е. все контекстно-свободные грамматики являются и контекстно-зависимыми, все контекстно-зависимые грамматики являются грамматиками общего вида и т.д. Кроме того, можно показать, что существуют языки, принадлежащие к типу i, но не к типу $i+1$. Например, язык G_3 является контекстно-зависимым, но не контекстно-свободным, т.е. не существует контекстно-свободной грамматики, порождающий этот язык. С другой стороны, некоторые нерегулярные грамматики могут порождать регулярные языки (например, грамматика G_1 – нерегулярная, но порождаемый ею язык регулярен, т.к. эквивалентная G_1 грамматика G_2 регулярна). Наконец, отметим, что определение контекстно-зависимой грамматики запрещает использование правил вида $A \to \varepsilon$. Это сделано для того, чтобы алгоритм, определяющий принадлежность цепочки языку, не мог бы зациклиться.

Распознаватели для различных классов грамматик

- Каждому классу языков соответствует эквивалентный класс распознавателей:
  - Язык, задаваемый грамматикой без ограничений, определяется машиной Тьюринга
  - Контекстно-зависимые языки определяются линейно ограниченными автоматами
  - Контекстно-свободные языки определяются автоматом с магазинной памятью
  - Праволинейные языки определяются конечным автоматом

Распознаватели для различных классов грамматик

Под *распознавателем* мы будем понимать обобщенный алгоритм, позволяющий определить некоторое множество (в нашем случае – язык) и использующий в своей работе следующие компоненты: входную ленту, управляющее устройство с конечной памятью и дополнительную рабочую память. Обычно считается, что управляющее устройство может только читать информацию, записанную на входной ленте (чтение производится с помощью входной головки, указывающей на текущий символ) и продвигаться по ней вперед и, возможно, назад. Распознаватель также может изменять состояние памяти, которая может быть организована как конечный линейный список ячеек или как стек (в русской литературе называемый также магазином). В качестве примеров распознавателей можно назвать машину Тьюринга, конечные и магазинные автоматы, которые должны быть известны студентам из предыдущих курсов.

Язык определяется путем задания некоторого множества допустимых заключительных состояний распознавателя: если цепочка, поданная на входную ленту, позволяет распознавателю выполнить последовательность шагов и остановиться в заключительном состоянии, то цепочка принадлежит языку.

Оказывается, каждому классу грамматик из иерархии Хомского соответствует класс распознавателей, определяющий тот же класс языков:

- Язык L праволинейный тогда и только тогда, когда он определяется (односторонним детерминированным) конечным автоматом
- Язык L контекстно-свободный тогда и только тогда, когда он определяется (односторонним недетерминированным) автоматом с магазинной памятью
- Язык L контекстно-зависимый тогда и только тогда, когда он определяется (двусторонним недетерминированным) автоматом с магазинной памятью
- Язык L рекурсивно перечислимый тогда и только тогда, когда он определяется машиной Тьюринга (этими понятиями мы оперировать не будем; формально они определяются в курсе "Вычислимость" или в базовом курсе "Компьютерные науки").

Дальнейший материал лекции будет посвящен определению свойств распознавателей, упомянутых в этих утверждениях, и обсуждению их применимости на практике.

Конечные автоматы

- Конечный автомат – это пятерка:

$M = (Q, \Sigma, \delta, q_0, F)$, где

1. Q – конечное множество состояний
2. Σ – конечное множестводопустимых входных символов
3. δ – функция перехода
4. q_0 из Q – начальное состояние
5. F – множество заключительных состояний

Конечные автоматы

Конечный автомат – это один из самых простых механизмов, используемых при работе с языками. У этого распознавателя есть только фиксированный набор ячеек памяти, а управляющее устройство может только сдвигаться вправо по входной ленте и изменять состояния памяти. Основная часть конечного автомата – это *функция перехода*, определяющая возможные переходы для любого текущего состояния и любого входного символа. Подразумевается, что допускается возможность перехода сразу в несколько различных состояний автомата, т.е. управляющее устройство распознавателя может быть и недетерминированным. Недетерминированность надо понимать следующим образом: если возможно сразу несколько переходов из данного состояния, то создается несколько копий нашего автомата, по одному на каждое новое состояние. Цепочка считается принадлежащей языку, если хотя бы одна из последовательностей шагов завершается в заключительном состоянии.

После такого краткого объяснения основных идей мы можем дать формальное определение конечного автомата.

Определение. Конечный автомат – это пятерка $M = (Q, \Sigma, \delta, q_0, F)$, где

- Q – конечное множество состояний
- Σ – конечное множество допустимых входных символов
- δ – отображение множества $Q \times \Sigma$ в множество P(Q), определяющее поведение управляющего устройства (эту функцию часто называют функцией переходов)
- $q_0 \in Q$ – начальное состояние управляющего устройства
- $F \subseteq Q$ – множество заключительных состояний

В принципе, для определения последующих действий конечного автомата достаточно знать текущее состояние управляющего устройства плюс последовательность все еще необработанных символов на входной ленте. Этот набор данных называется конфигурацией автомата.

Детерминированные конечные автоматы

- Автомат называется *детерминированным*, если множество δ(*q, a*) содержит не более одного состояния для любых *q, a*. Если δ(*q, a*) всегда содержит ровно одно состояние, то автомат называется *полностью определенным*.
- Цепочка w допускается автоматом M, если существует последовательность шагов, приводящая нас по этой цепочке в заключительное состояние автомата
- Язык распознается конечным автоматом, если им распознается каждое слово языка
- Удобная форма записи конечных автоматов – диаграммы переходов

Детерминированные конечные автоматы

Определим детерминированный конечный автомат как частный случай общего (недетерминированного) конечного автомата и введем некоторые дополнительные определения и соглашения, которые пригодятся нам в дальнейшем:

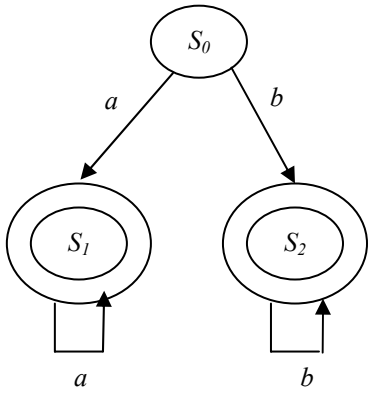
Определение. Автомат называется *детерминированным*, если множество δ(*q, a*) содержит не более одного состояния для любых *q, a*. Если δ(*q, a*) всегда содержит ровно одно состояние (т.е. не имеет неопределенных переходов), то автомат называется *полностью определенным*.

Определение. Слово w = $a_1 \ldots a_k$ над алфавитом Σ допускается конечным автоматом M = (Q, Σ, δ, q_0, F), если существует последовательность состояний $q_1, q_2, \ldots, q_n$ такая, что $q_1 = q_0$, $q_n \in F$ и для $\forall i \forall j : 1 \le i < n, 1 \le j < k \quad \delta(q_i, a_j) = q_{i+1}$.

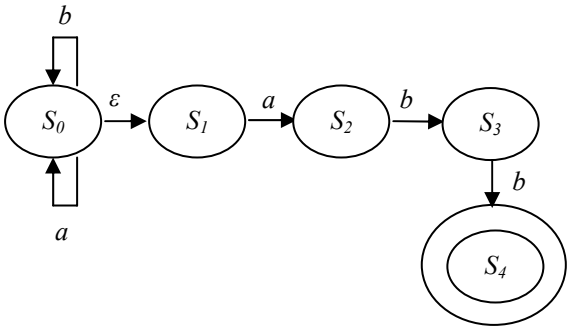
Определение. Язык L распознается конечным автоматом, если каждое слово языка L допускается этим конечным автоматом.

Обозначение. Конечные автоматы удобно иллюстрировать с помощью *диаграмм переходов*, см. примеры ниже (двойным кружком обозначены конечные состояния):

Автомат, распознающий язык (*aa*|bb**):



Автомат, распознающий язык *(a|b)*abb*:



Недетерминированные и конечные автоматы

- Любому недетерминированному автомату соответствует детерминированный автомат, определяющий тот же самый язык, причем известен метод конструирования эквивалентного конечного автомата
- Таким образом, классы языков, задаваемых недетерминированными и детерминированными конечными автоматами, совпадают
- Конечные автоматы – удобный формализм, так как их легко моделировать программно

Недетерминированные и конечные автоматы

Имеет место следующая **теорема**: если L = L(M) для некоторого недетерминированного конечного автомата M, то L = L(M') для некоторого детерминированного автомата M'.

Эта теорема доказывается конструктивным образом, т.е. путем указания общего алгоритма построения детерминированного автомата M', определяющего тот же язык, что и M. Пусть M = (Q, Σ, δ, q_0, F); тогда мы определим M' = (Q', Σ, δ', q_0', F') следующим образом:

- Q' совпадает с множеством состояний автомата M
- $q_0' = q_0$
- F' = $\{S \in Q \mid S \cap F \neq 0\}$
- δ'(S, a) = S' для всех $S \subseteq Q$, где S' = { p | δ(q, a) содержит p для некоторого $q \in S$ }

Можно показать, что M' задает тот же язык, что и M. Таким образом, классы языков, задаваемых детерминированными и недетерминированными конечными автоматами, полностью совпадают. Естественно, детерминированные конечные автоматы удобнее, и в дальнейшем мы будем иметь дело только с ними. Следующий набросок программы демонстрирует моделирование конечного автомата (предполагается, что входная лента заканчивается символом *end_of_file*):

```
q = q0;
c = GetChar();
while (c != eof) {
  q = move (q, c);
  c = GetChar();
}
if (q is in F) return "yes";
else return "no";
```

Эквивалентность конечных автоматов

- Два детерминированных автомата называются эквивалентными, если они распознают один и тот же язык
- Алгоритм проверки эквивалентности конечных автоматов

Эквивалентность конечных автоматов

Определение. Два автомата $M_1 = (Q_1, \Sigma, \delta_1, q_{10}, F_1)$ и $M_2 = (Q_2, \Sigma, \delta_2, q_{20}, F_2)$ называются эквивалентными, если они распознают один и тот же язык над алфавитом Σ.

Определение. Два состояния s_i и s_j называются эквивалентными, если $\forall x \in \Sigma^*$ верно, что $\delta(q_i, x) \in F \Leftrightarrow \delta(q_j, x) \in F$. Очевидно, что если два состояния s_i и s_j эквивалентны, то $\forall a \in \Sigma$ состояния $\delta(s_i, a)$ и $\delta(s_j, a)$ также эквивалентны.

Кроме того, так как в детерминированном конечном автомате переход $\delta(q,\varepsilon)$ может возникнуть только для конечного состояния q, то никакое заключительное состояние не может быть эквивалентно незаключительному состоянию. Таким образом, если мы предположим, что начальные состояния автоматов эквивалентны, то мы можем получить и другие пары эквивалентных состояний. Если в одну из таких пар попадет заключительное состояние вместе с незаключительным, то s_i и s_j неэквивалентны. Напишем алгоритм разбиения множества состояний на классы эквивалентности:

```
Добавить  (q10, q20) в Список;
Список = 0; /* Множество эквивалентных множеств */
for each (q in Q1+Q2) { Добавить {s} в Список; }
while (есть пара (qi, qj), входящая в Список) {
   Удалить пару (qi, qj) из Списка;
   Пусть A и A′  - такие множества, что qi ∈ A  и  qj ∈ A′;
   if (A != A′) {
      A = A + A′;
      for (a from Σ) { Добавить (δ(qi, a), δ(qj, a)) в Список; }
   }
}
```

Таким образом, мы получим разбиение множества $Q_1 \cup Q_2$ на множества эквивалентных состояний, если q_{10} и q_{20} – эквивалентны. Теперь осталось проверить, что никакое из этих множеств не содержит заключительное и незаключительное состояния. Если это верно, то автоматы эквивалентны.

Минимизация конечного автомата

- Как найти автомат, эквивалентный данному, с минимальным числом состояний?
- Алгоритм минимизации конечного автомата выглядит так:
  - Вначале мы удаляем все недостижимые состояния
  - Затем разбиваем множество всех достижимых состояний на классы эквивалентности неразличимых состояний
  - Из каждого класса эквивалентности мы берем только по одному представителю

Минимизация конечного автомата

Учитывая, что мы можем определять эквивалентные автоматы, хотелось бы научиться находить наименьший среди них. Задача нахождения наименьшего конечного автомата, распознающего данный язык, также разрешима, причем тоже конструктивным образом. Для описания процесса нахождения наименьшего конечного автомата, введем следующее определение.

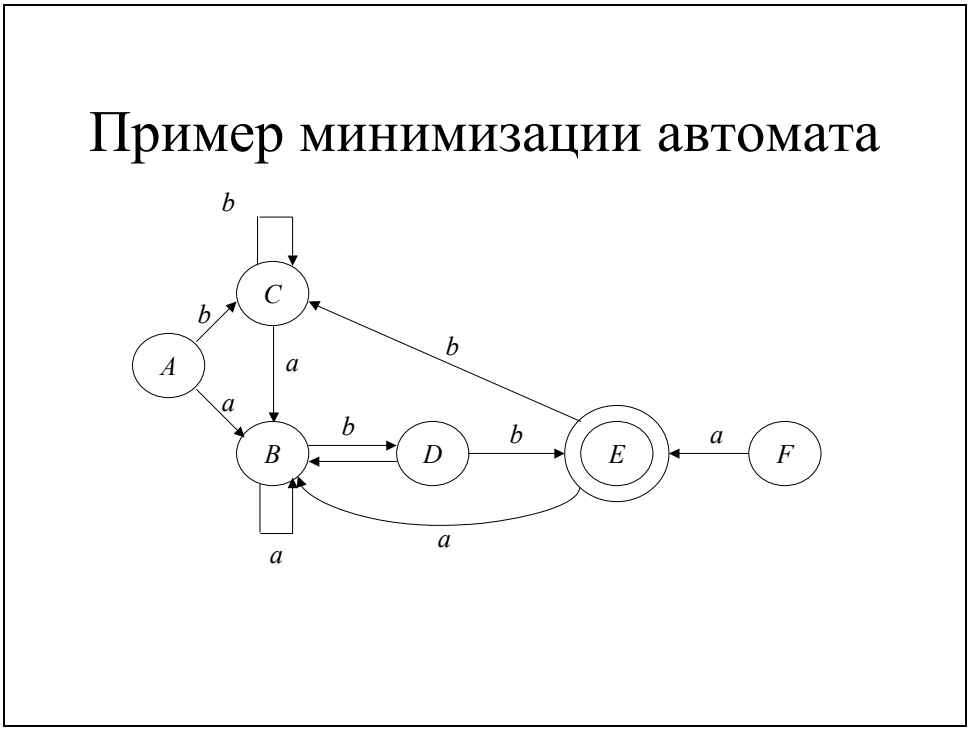
Определение. Будем говорить, что строка w *различает* состояния s и t, если существует такая входная цепочка w, что $\delta(s,w)$ является заключительным состоянием, а состояние $\delta(t,w)$ не является заключительным состоянием, или, наоборот, $\delta(s,w)$ не является заключительным состоянием, а состояние $\delta(t,w)$ является заключительным состоянием.

Очевидно, что если некоторая строка w различает состояния t_1 и t_2 такие, что $t_1 = \delta(q_1, a)$ и $t_2 = \delta(q_2, a)$, то строка aw различает состояния q_1 и q_2.

Теперь перейдем к описанию процесса минимизации конечного автомата. Мы начнем с поиска и удаления всех недостижимых состояний. Затем мы должны найти такое разбиение множества состояний автомата, чтобы каждое подмножество содержало неразличимые состояния, т.е. если s и t принадлежат некоторому подмножеству, то для всех a из Σ $\delta(s,a)$ и $\delta(t,a)$ также принадлежат этому подмножеству.

Для этого мы разобьем множество состояний на два подмножества: F и S–F. В дальнейшем, мы попытаемся разбить каждое из подмножеств, соблюдая указанное выше условие. Если возникает ситуация, при которой мы не можем разбить никакое множество состояний, то мы заканчиваем процесс разбиения. В результате мы получим некоторый набор множеств состояний $S_1,\ldots,S_k$. Каждое из S_i содержит только неразличимые состояния. Наконец, внесем в множество состояний минимизированного автомата по одному представителю каждого из множеств S_i . На этом процесс завершается.

На следующем слайде мы приведем пример минимизации автомата.



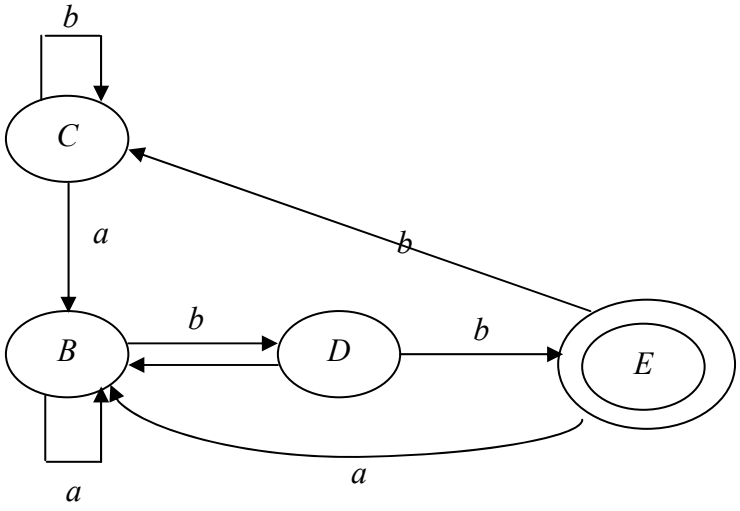
Пример минимизации конечного автомата

Рассмотрим процесс минимизации автомата, представленного на слайде. Согласно алгоритму, вначале мы произведем удаление недостижимых состояний – в нашем примере состояние F очевидно недостижимо и потому не попадет в минимизированный автомат.

Затем мы произведем разбиение множества состояний автомата на классы эквивалентности. Укажем такую последовательность разбиений:

1. E, ABCD
2. E, ABC, D, так как $\delta(D,b)$ = E.
3. E, AC, B, D, так как $\delta(B,b)$ = D.

Таким образом, состояния A и C неразличимы. Поэтому получаем следующий автомат:



Эквивалентность праволинейных грамматик и конечных автоматов

- Любой конечно-автоматный язык может быть определен праволинейной грамматикой и наоборот, т.е. классы языков, определяемых этими формализмами, эквивалентны
- Для класса языков, задаваемых праволинейными грамматиками, можно проверить следующие свойства:
  - Эквивалентность двух языков
  - Пустоту определяемого языка
  - Принадлежит ли входная цепочка заданной грамматике
- К сожалению, праволинейные грамматики позволяют задать только весьма ограниченный класс языков

Эквивалентность праволинейных грамматик и конечных автоматов

Как мы упоминалось выше, любой конечно-автоматный язык может быть определен праволинейной грамматикой, и наоборот, так что классы языков, определяемых этими формализмами, эквивалентны. Так, для конструирования праволинейной грамматики, соответствующей данному конечному автомату, достаточно включить в грамматику правила вида $q \to ar$ для всех переходов вида $\delta(q,a) = r$ и правила вида $p \to \varepsilon$ для всех заключительных состояний p.

Таким образом, класс языков, задаваемых праволинейными грамматиками, очень удобен в задачах компиляции, т.к. ему соответствует простой распознаватель (конечные автоматы), для которого алгоритмически разрешимы такие проблемы, как эквивалентность двух языков, пустота определяемого языка и проверка входной цепочки на принадлежность данному языку. Многие "локальные" свойства языков программирования, такие как константы, слова языка и строки, могут быть определены с помощью праволинейных грамматик – например, в следующей лекции мы покажем как этот формализм может быть использован в задачах лексического анализа.

Однако, класс языков, задаваемых праволинейными грамматиками, слишком узок для описания многих свойств современных языков программирования. Например, в большинстве языков программирования возникает потребность в согласовании подобных скобок и разделителей, таких, как begin-end, (), [], {}. Можно промоделировать подобное согласование с помощью "правильного скобочного языка", в котором алфавит состоит из символов '(' и ')', а количество открывающих скобок совпадает с количеством закрывающих и число закрывающих скобок никогда не превышает число встреченных открывающих скобок. Можно показать, что не существует праволинейной грамматики, описывающий данный язык, но зато его легко записать с помощью следующей контекстно-свободной грамматики: $S \to (S)$, $S \to SS$, $S \to \varepsilon$. По этой причине контекстно-свободные грамматики получили значительно большее распространение в компиляции. С их помощью можно задать очень большую часть синтаксиса языков программирования.

Свойства контекстно-свободных грамматик

- Процесс синтаксического анализа программы можно рассматривать как определение принадлежности некоторой цепочки данной контекстно-свободной (КС-) грамматике
- Существует проблема неоднозначности грамматик
- Проблема учета приоритетов операций
- Для разрешения этих и других проблем прибегают к преобразованию грамматик

Свойства контекстно-свободных грамматик

Весь процесс синтаксическо анализа можно упрощенно рассматривать как определение принадлежности входной цепочки заданной контекстно-свободной грамматике (в дальнейшем, мы будем использовать сокрещение "КС-грамматика"). Однако надо оговориться, что такая формулировка сильно упрощена, так как процесс вывода цепочки по грамматике нетривиален и к тому же усложняется тем фактом, что далеко не все свойства языков программирования могут быть описаны КС-грамматиками.

Кроме того, КС-грамматики зачастую страдают от проблемы неоднозначности. Грамматика называется *неоднозначной (ambiguous grammar)*, если существует, по крайней мере, одна выводимая в этой грамматике цепочка, для которой существует более одного вывода. Например, можно рассмотреть язык G_0, описывающий простые арифметические выражения со сложением и умножением, и задаваемый КС-грамматикой следующим набором правил: $E \rightarrow E+E \mid E*E \mid (E) \mid a$. Эта грамматика неоднозначна, так как в ней существует два различных вывода для выражения вида $a+a+a$ или $a*a+a$ (левосторонний и правосторонний вывод).

В данном случае проблему неоднозначности можно решить путем преобразования грамматики к эквивалентной грамматике G_1: $E \rightarrow E+T \mid E*T \mid T,\ T \rightarrow (E) \mid a$. Но и у этой грамматики есть серьезный недостаток: операции + и * имеют один и тот же приоритет, и потому выражение $a+a*a$ будет трактоваться как $(a+a)*a$. Для того, чтобы получить правильный приоритет операций, необходимо продолжить преобразование грамматик и перейти к следующему набору правил: $E \rightarrow E+T \mid T,\ T \rightarrow T*F \mid F,\ F \rightarrow (E) \mid a$. Более подробно проблемы, связанные с неоднозначными грамматиками, будут рассмотрены в лекции 8.

О преобразовании грамматик

- К сожалению, не существует общего механизма приведения КС-грамматик к произвольному виду
- Однако известно, что любая КС-грамматика может быть приведена к нормальному виду Хомского
- Другой стандартный вид КС-грамматик – это нормальная форма Грейбах

О преобразовании грамматик

Как мы уже видели в предыдущем примере, иногда может возникнуть потребность в преобразовании КС-грамматик.

К сожалению, не существует общего алгоритмического метода приведения КС-грамматик к произвольному виду, как не существует и алгоритма, определяющего равенство языков, задаваемых двумя КС-грамматиками. Однако известен целый ряд преобразований, таких как устранение бесполезных символов и устранение цепных правил, которые позволяют привести грамматику к более удобной форме.

Кроме того, известно, что любая КС-грамматика может быть приведена к *нормальному виду Хомского*, в котором все правила имеют один из следующих видов:

- $A \to BC$, где A, B и C – нетерминалы
- $A \to a$, где a – терминал
- $S \to \varepsilon$, при условии, что символ ε принадлежит языку, а нетерминал S не встречается в правых частях правил

Другим широко распространенным стандартным видом КС-грамматик является *нормальная форма Грейбах*, в которой все правые части правил начинаются с терминалов.

Перейдем к рассмотрению класса распознавателей, соответствующих классу языков, задаваемых КС-грамматиками – к магазинным автоматам.

Магазинные автоматы

- МП-автомат определяется как семерка

P = (Q, Σ, Γ, δ, q_0, Z_0, F), где:

Q – конечное множество состояний

Σ – конечный входной алфавит

Γ – конечный алфавит магазинных символов

δ – функция перехода

q_0 из Q – начальное состояние

Z_0 из Γ – начальный символ магазина

F – множество заключительных состояний

- Известно, что МП-автоматы задают тот же класс языков, что и КС-грамматики

Магазинные автоматы

Магазинные автоматы, известные также как автоматы с магазинной памятью или как МП-автоматы, формально определяются следующим образом.

Определение. МП-автомат – это семерка P = (Q, Σ, Γ, δ, q_0, Z_0, F), где:

- Q – конечное множество состояний
- Σ – конечный входной алфавит
- Γ – конечный алфавит магазинных символов
- δ – функция перехода, отображение множества $Q \times (\Sigma \cup \{e\}) \times \Gamma$ в множество конечных подмножеств множества $Q \times \Gamma^*$
- $q_0 \in Q$ – начальное состояние управляющего устройства
- $Z_0 \in \Gamma$ – символ, находящийся в магазине в начальный момент (начальный символ)
- $F \subseteq Q$ – множество заключительных состояний.

Определение. Конфигурацией МП-автомата P называется тройка $(q, \omega, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, где q – текущее состояние управляющего устройства, ω – неиспользованная часть входной цепочки (если $\omega=\varepsilon$, то считается, что вся входная цепочка прочитана), α – содержимое магазина (самый левый символ цепочки α считается верхним символом магазина; если $\alpha=\varepsilon$, то магазин считается пустым).

На каждом шаге работы МП-автомат может либо занести что-то в магазин, либо снять какие-то значения с его вершины. Отметим, что МП-автомат может продолжать работать в случае окончания входной цепочки, но не может продолжать работу в случае опустошения магазина.

Класс языков, распознаваемых МП-автоматами, в точности совпадает с классом языком, задаваемых КС-грамматиками (мы этого доказывать не будем).

Пример магазинного автомата

- МП-автомат, распознающий язык $\{0^n1^n\}$:

$\delta(q_0, 0, Z) = \{(q_1, 0Z)\}$
$\delta(q_1, 0, 0) = \{(q_1, 00)\}$
$\delta(q_1, 1, 0) = \{(q_2, \varepsilon)\}$
$\delta(q_2, 1, 0) = \{(q_2, \varepsilon)\}$
$\delta(q_0, \varepsilon, Z) = \{(q_0, \varepsilon)\}$

Пример магазинного автомата

Рассмотрим магазинный автомат, распознающий язык $\{0^n1^n \mid n \geq 0\}$.
Пусть $P = (\{q_0, q_1, q_2\}, \{0,1\}, \{Z,0\}, \delta, q_0, Z, \{q_0\})$, где:

$\delta(q_0, 0, Z) = \{(q_1, 0Z)\}$
$\delta(q_1, 0, 0) = \{(q_1, 00)\}$
$\delta(q_1, 1, 0) = \{(q_2, \varepsilon)\}$
$\delta(q_2, 1, 0) = \{(q_2, \varepsilon)\}$
$\delta(q_2, \varepsilon, Z) = \{(q_0, \varepsilon)\}$

Работа автомата заключается в копировании в магазин начальных нулей из входной цепочки и последующем устранении по одному нулю из магазина на каждую прочитанную единицу.

Детерминированные МП-автоматы

- МП-автоматы недетерминированны. Поэтому мы рассмотрим класс детерминированных МП-автоматов, которые в каждой конфигурации могут сделать не более одного такта
- Детерминированные МП-автоматы очень полезны в задачах компиляции
- К сожалению, детерминированные МП-автоматы определяют более слабый класс языков, чем КС-грамматики.

Детерминированные МП-автоматы

МП-автоматы обладают одним существенным недостатком – они недетерминированны по своей природе. На практике нам хотелось бы иметь дело с детерминированными автоматами, в которых в каждой конфигурации возможно не более одного такта:

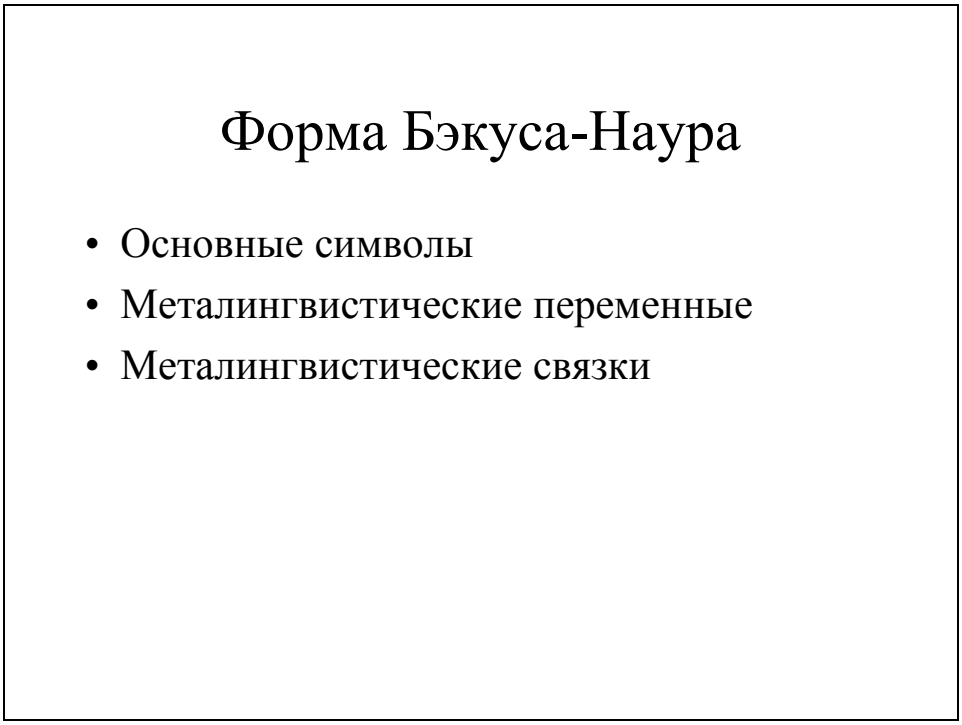
Определение. МП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ называется *детерминированным*, если для каждых $q \in Q$ и $Z \in \Gamma$ верно одно из следующих утверждений:

- $\delta(q, a, Z)$ содержит не более одного элемента для каждого $a \in \Sigma$ и $\delta(q, \varepsilon, Z) = \varnothing$
- $\delta(q, a, Z) = \varnothing$ для всех $a \in \Sigma$ и $\delta(q, \varepsilon, Z)$ содержит не более одного элемента

К сожалению, детерминированные МП-автоматы описывают только подмножество всего класса КС-языков – это подмножество называется *детерминированными КС-языками*. Этот класс языков называют также LR(k)-грамматиками, так как они могут быть однозначно разобраны путем просмотра цепочки слева направо с заглядыванием вперед не более, чем на k символов.

Детерминированные КС-языки, как и весь класс КС-языков, позволяют проверить пустоту определяемого ими языка или принадлежность заданной цепочки языку. Однако, детерминированные КС-языки не обладают некоторыми важными теоретико-множественными свойствами, например, детерминированные КС-языки не замкнуты относительно пересечения, дополнения или объединения (общий класс КС-языков также не замкнут относительно пересечения и дополнения, но замкнут относительно объединения).

Тем не менее, с точки зрения методов компиляции, класс LR(k)-грамматик чрезвычайно важен, так как на нем и некоторых его разновидностях основаны большинство современных средств синтаксического разбора. Мы рассмотрим LR(k)-грамматики в отдельной лекции.



Форма Бэкуса-Наура

Перейдем теперь к рассмотрению наиболее распространенных способов задания синтаксиса современных языков программирования. Естественно, наиболее часто используется какой-то вид формальных грамматик. Однако, с помощью формальной грамматики определяется только контекстно-независимая составляющая языка. Поэтому для реальных языков программирования мы не можем в общем случае сказать, что полученная с помощью такой грамматики цепочка терминалов является синтаксически правильной программой, так как правильная программа должна удовлетворять еще контекстным условиям.

Один из наиболее распространенных способов описания синтаксиса языка – это *форма Бэкуса-Наура* (Backus J.W., Naur P.). Этот способ был разработан для описания Алгола-60, однако, в дальнейшем он использован для многих других языков. При записи грамматики в форме Бэкуса-Наура используются два типа объектов:

- основные символы (или терминальные символы, в частности, ключевые слова)
- металингвистические переменные (или нетерминальные символы), значениями которых являются цепочки основных символов описываемого языка. Металингвистические переменные изображаются словами (русскими или английскими), заключенными в угловые скобки (<...>)
- металингвистические связки (::=, |)

Пример:
<целое> ::= <целое без знака> | + <целое без знака> | - <целое без знака>
<целое без знака> ::= <цифра> | <целое без знака> <цифра>
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Форма Бэкуса-Наура не позволяет задавать контекстные условия. При использовании формы Бэкуса-Наура контекстные условия задаются в словесной форме.

Расширенная форма Бэкуса-Наура

- Расширенная форма Бэкуса-Наура, использованная Виртом
- В 1981 году Британский институт стандартов (British Standards Institute) опубликовал стандарт EBNF (BSI).

Расширенная форма Бэкуса-Наура

При определении синтаксиса языков Pascal и Modula-2 Вирт использовал *расширенную форму Бэкуса-Наура* (EBNF):

- Нетерминалы записываются как отдельные слова
- Терминалы записываются в кавычках, например, “BEGIN”
- Вертикальная черта (|), как и прежде, используется для определения альтернатив
- Круглые скобки используются для группировки
- Квадратные скобки используются для определения возможного вхождения символа или группы символов
- Фигурные скобки используются для определения возможного повторения символа или группы символов
- Символ равенства используется вместо символа ::=
- Символ точка используется для обозначения конца правила
- Комментарии заключаются между символами (* … *)
- ε эквивалентно [ ]

Пример.

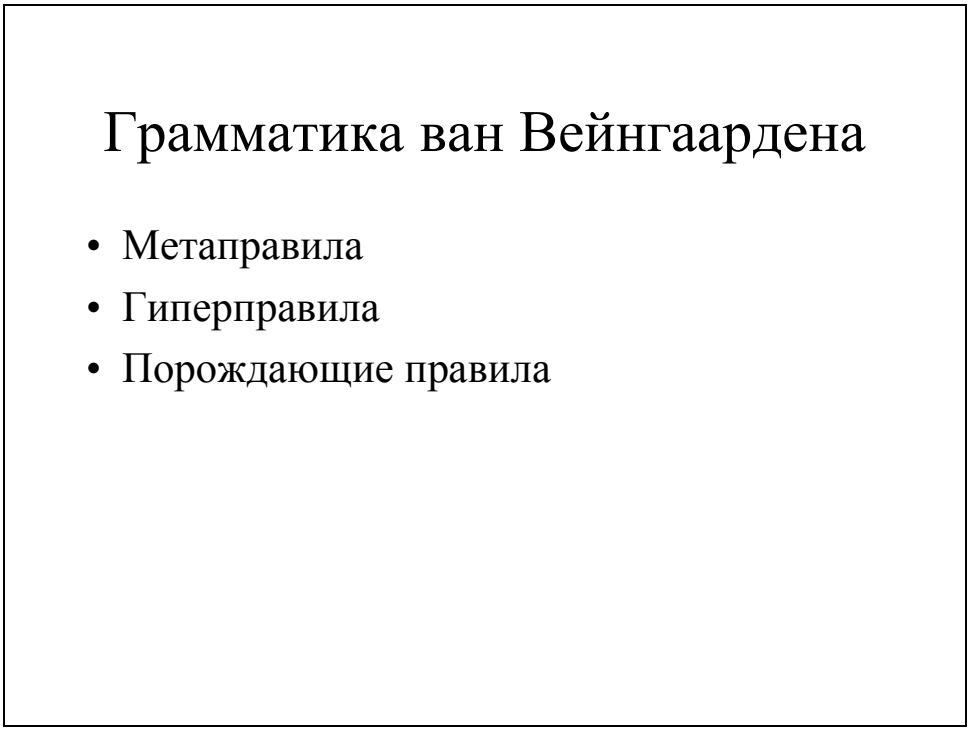
Integer = Sign UnsignedInteger.
UnsignedInteger = digit {digit}.
Sign = ["+"|"–"].
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

В 1981 году Британский институт стандартов (British Standards Institute) опубликовал стандарт EBNF. BSI стандарт получился более наглядным, чем расширенная форма Бэкуса-Наура, предложенная Виртом.

- Элементы правил разделяются запятыми
- Правила заканчиваются точкой с запятой
- Пробелы не являются значащими

Пример.

Constant Declaration = "CONST",
Constant Identifier , "=", Constant Expression, ";",
{Constant Identifier, "=", Constant Expression, ";"};
Constant Identifier = identifier;
Constant Expression = Expression;



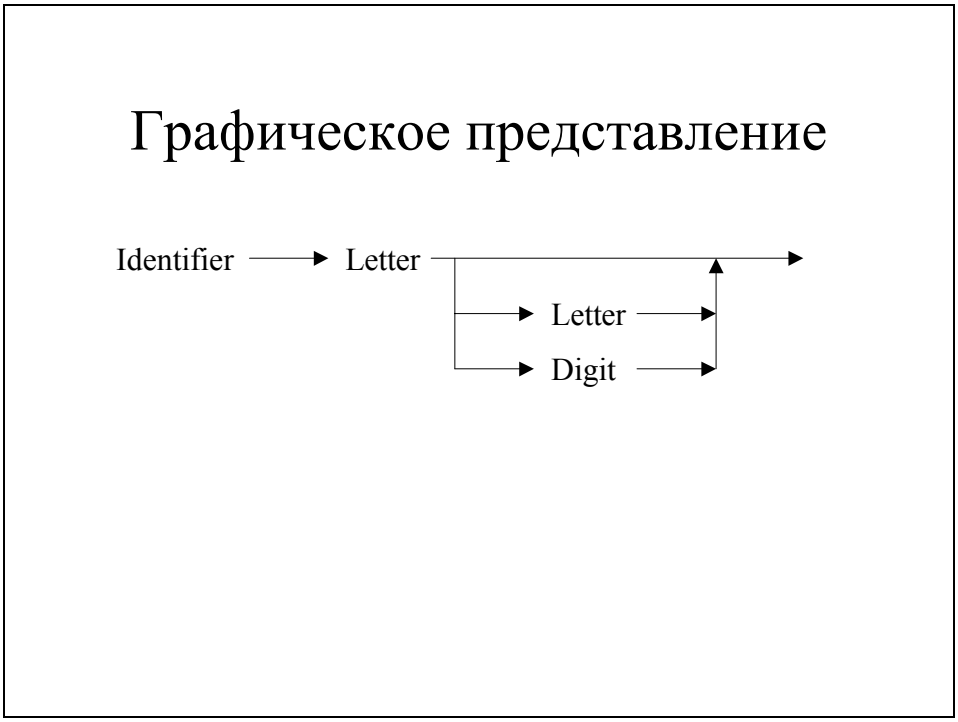
Грамматика ван Вейнгаардена

В конце 60-х годов при разработке языка Algol 68 была предложена новая форма определения языка, позволяющая формулировать не только контекстно-свободный синтаксис, но и контекстные условия. Такой способ определения языка получил название *грамматики ван Вейнгаардена*. Основная идея заключается в том, что это так называемая двухуровневая грамматика. Синтаксис действует следующим образом: заданы множества «гиперправил» и «метаправил», из которых могут выводиться «порождающие правила». Гиперправила – это просто «заготовки» правил. С помощью метаправил задается контекстная составляющая языка.

Пример. Рассмотрим правило, определяющее присваивание.
REF to MODE NEST assignation:
REF to MODE NEST destination, becomes token,
MODE NEST source.

Большими буквами в этом правиле написаны метасимволы. Метасимволы REF и MODE задают тип (или, как принято говорить, вид) конструкции, причем метасимвол REF означает, что получатель присваивания (destination) должен быть переменной типа MODE. Метасимвол NEST задает контекст конструкции. Обратите внимание, что контекст получателя и источника присваивания совпадают. Из этого правила может быть выведено, например, порождающее правило:

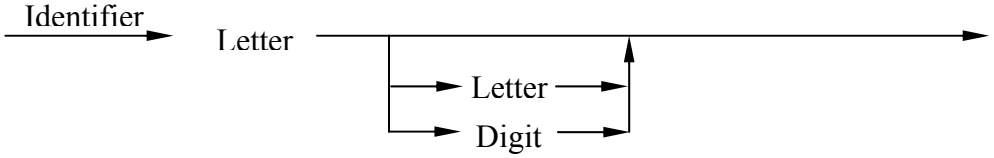
reference to real assignation:
reference to real destination, becomes token,
real source



Графическое представление

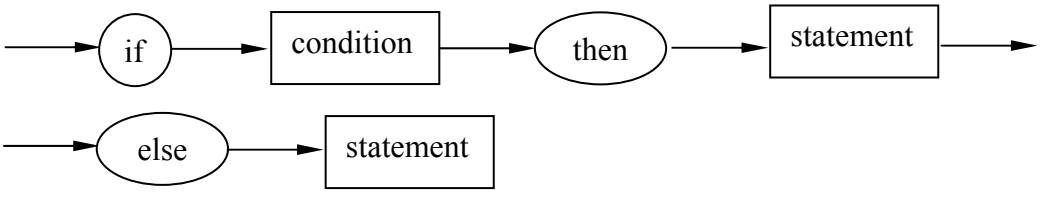
Совершенно иной способ представления синтаксиса – это *графическое представление*. Такое представление известно как синтаксические диаграммы (syntax diagrams) или синтаксические схемы (syntax charts). Они использовались для определения синтаксиса языков Pascal, Modula-2. Они имеют форму блок-схем (flow diagram).

Пример.



Часто вместо таких синтаксических диаграмм используются другие, в которых терминалы записываются в кружочках, а нетерминалы – в прямоугольниках. Такие синтаксические диаграммы, действительно, похожи на блок-схемы.

Пример.



Литература к лекции

- А. Ахо, Дж. Ульман "Теория синтаксического анализа, перевода и компиляции", Т.1 "Синтаксический анализ", М.: Мир, 1978
- Р. Хантер "Проектирование и конструирование компиляторов", ФиС, 1984

Литература к лекции

- А. Ахо, Дж. Ульман "Теория синтаксического анализа, перевода и компиляции", Т.1 "Синтаксический анализ", М.: Мир, 1978
- Р. Хантер "Проектирование и конструирование компиляторов", ФиС, 1984