Прикладное применение теории автоматов

Лекция 01 – Введение

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2023

2023.02.27

Оглавление

Отличительной особенностью объектно-ориентированного кода является то, что он может сделать маленькие и простые задачи похожими на большие и сложные.

Альтернативы ОО-программированию

В качестве методологии программирования ОО-программирование конкурирует с несколькими другими:

- двухуровневая конструкция с использованием языка сценариев плюс язык низкого уровня;
- определение абстрактной машины или специализированного языка для задачи;
- рассматривать входные данные как язык и структурировать приложение в виде компилятора;
- виртуальные устройства и использование LAMP[1] стека;
- безопасное программирование.

Методология программирования на двух языках

Использование в одном проекте языка сценариев, например такого как TCL, Lua, Julia, и компилируемого языка, такого как C отделяет программирование в большом (склеивающий язык) от программирования в малом (компонентное программирование).

Для многих задач подход «язык сценариев + компилируемый язык» является лучшей парадигмой разработки программного обеспечения, поскольку навязывает программистам C дисциплину разработки установленную в сообществе разработчиков сценариев-интерпретаторов. Библиотеки, используемые интерпретатором, обычно очень высокого качества и могут служит примером того, как все будет сделано, а также для предотвращения «повторного изобретения колеса». Программисты, как правило, учатся на примерах, и код даже простого интерпретатора, такого как AWK или gawk, — отличная школа.

---

1) Linux, Apache, MySQL, PHP.

10-й закон программирования Гринспуна

«Любая достаточно сложная ОО-программа, написанная на Java, C++ или другом ОО-языке, содержит специальную, подверженную ошибкам и медленную реализацию половины интерпретатора языка сценариев.»

Определение абстрактной машины и специализированного языка

В основе лежит идея структурирования приложения как абстрактной машины с четко определенными примитивами (кодами операций), которая управляется с помощью специализированного языка.

Специализированный язык не обязан создавать объектный код — более эффективным подходом является компиляция на язык более низкого уровня, такой как C, C ++ или Java.

В этом случае поддерживаемое приложение может быть разделено на две отдельные части:

- обслуживание кодовой базы более высокого уровня;
- обслуживание абстрактной машины, реализующей язык этого самого более высокого уровня и связанную инфраструктуру времени выполнения.

Стоимость программирования сильно зависит от уровня используемого языка, поэтому использование языка более высокого уровня позволяет значительно снизить стоимость разработки.

Входные данные как язык, а приложение – компилятор

Данная технология сильно недооценена и мало используется в современной разработке ПО.

Она позволяет обнаружить некоторые ошибки высокого уровня на синтаксическом уровне, что невозможно в ОО.

Задача может быть структурирована в компиляторо-подобную форму с выделенным лексическим анализатором, синтаксическим анализатором и кодогенерирующими компонентами.

Многопроходная компиляция с промежуточным представлением, доступным для записи на диск, во многих случаях является отличным инструментом для решения сложной проблемы.

Если формально определены промежуточные представления между различными проходами, они также могут быть проанализированы на правильность.

Также отделение фаз обработки друг от друга значительно упрощает отладку.

В качестве промежуточного языка представления в последнее время стало модно использовать XML, хотя в подавляющем большинстве случаем это излишнее.

История возникновения трансляторов

Машинные коды

@ трудоемкость программирования непосредственно в машинных кодах
@ каждый процессор имеет свою собственную систему команд
==> программа, написанная для одного процессора, не переносима на другой в общем случае
==> переход на новую процессорную архитектуру требует переобучения персонала
==> псевдокоды (Д3-28).

Первые языки Assembler'а, использующие мнемонические обозначения команд

@ программы с мнемоническими обозначениями значительно нагляднее, меньше ошибок
@ язык ассемблера по-прежнему низкоуровневый и ориентирован на архитектуру конкретного процессора
==>необходимость в программах-переводчиках с языка Assembler'а, а в дальнейшем и с языков более высокого уровня, в машинные коды.

Эти программы получили название ***трансляторов*** (translate – переводить).

> ***Трансляция программы*** — преобразование программы, представленной на одном из языков программирования, в программу, написанную на другом языке.
>
> Язык, на котором представлена входная программа, называется ***исходным языком***, а сама программа — исходным кодом. Выходной язык называется ***целевым языком***.

Фортран (англ. Fortran) — первый язык программирования высокого уровня, получивший практическое применение, имеющий транслятор и испытавший дальнейшее развитие. Создан в период с 1954 по 1957 год группой программистов под руководством Джона Бэкуса в корпорации IBM.

Фортран широко используется до сих пор для научных и инженерных вычислений (MPI).

Основные преимущества современного Фортрана — большое количество написанных на нём программ и библиотек подпрограмм и высокая производительность.

Классификация трансляторов

Компиляторы
Ассемблеры
Кроссс-компиляторы
Компиляторы компиляторов
Динамические компиляторы
Интерпретаторы
Детрансляторы
Дизассемблеры
Препроцессоры
Макрогенераторы

Трансляторы – это программные средства, выполняющие преобразование программ, представленных на одном языке, в эквивалентную ей программу на другом языке.

Компилятор (**compiler**) переводит программу с исходного языка на язык более низкого уровня. На входе компиляторы получают исходный текст программы, а на выходе выдают готовую программу в машинном, объектном или ином промежуточном коде.

Асемблер — компилятор с языка Assembler'a.

Кросс-компилятор — обрабатывает исходный код программы на выч. платформе A, формируя объектный код для платформы B.

Компилятор компиляторов на вход получает описание некоторого языка $\mathcal{L}$ в терминах формальных грамматик, а на выходе формирует текст программы-компилятора на языке высокого уровня, чаще всего на C, которая позволяет выполнять компиляцию программ на языке $\mathcal{L}$. (LEX, FLEX, YACC, Bison)

Интерпретаторы – это компиляторы, которые не формируют готовой программы, а выполняют исходный текст программы по частям (TCL/TK, bash, awk, python)/

Препроцессор – это транслятор макрорасширений языка, который переводит их в программу на входном языке (cpp).

Макрогенератор — препроцессор ассемблера (nasm).

Детранслятор преобразует программу с языка более низкого уровня к языку более высокого уровня.

Дизассемблер — детранслятор на язык ассемблера (nasm – ndisasm, edb, objdump).

Этапы трансляции

- препроцессор;
- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- распределение памяти;
- оптимизация;
- генерация объектного кода.

Препроцессор — обработка макросов, включение файлов, языковые расширения

Лексический анализ — проверка правильности написания основных элементарных конструкций языка (лексем) – констант, идентификаторов, ключевых слов

Синтаксический анализ — проверка правильности синтаксических конструкций, сформированных из лексем

Семантический анализ — проверка контекстных зависимостей, проверка соблюдения правил объявления данных до их использования и иных подобных правил

Распределение памяти — назначение адресов для данных программы

Оптимизация — обнаружение неиспользуемых данных, выделение инвариантов, ...

Генерация кода (генерируется программа на выходном языке транслятора эквивалентная исходной).

Жизненный путь программы

Обычно программа проходит нескольких стадий:
- текст на алгоритмическом языке;
- объектный модуль;
- загрузочный модуль;
- бинарный образ в памяти.

Используемые программой *адреса памяти* в каждом конкретном случае могут быть представлены различными способами. Например, адреса в исходных текстах обычно *символические* (идентификаторы).

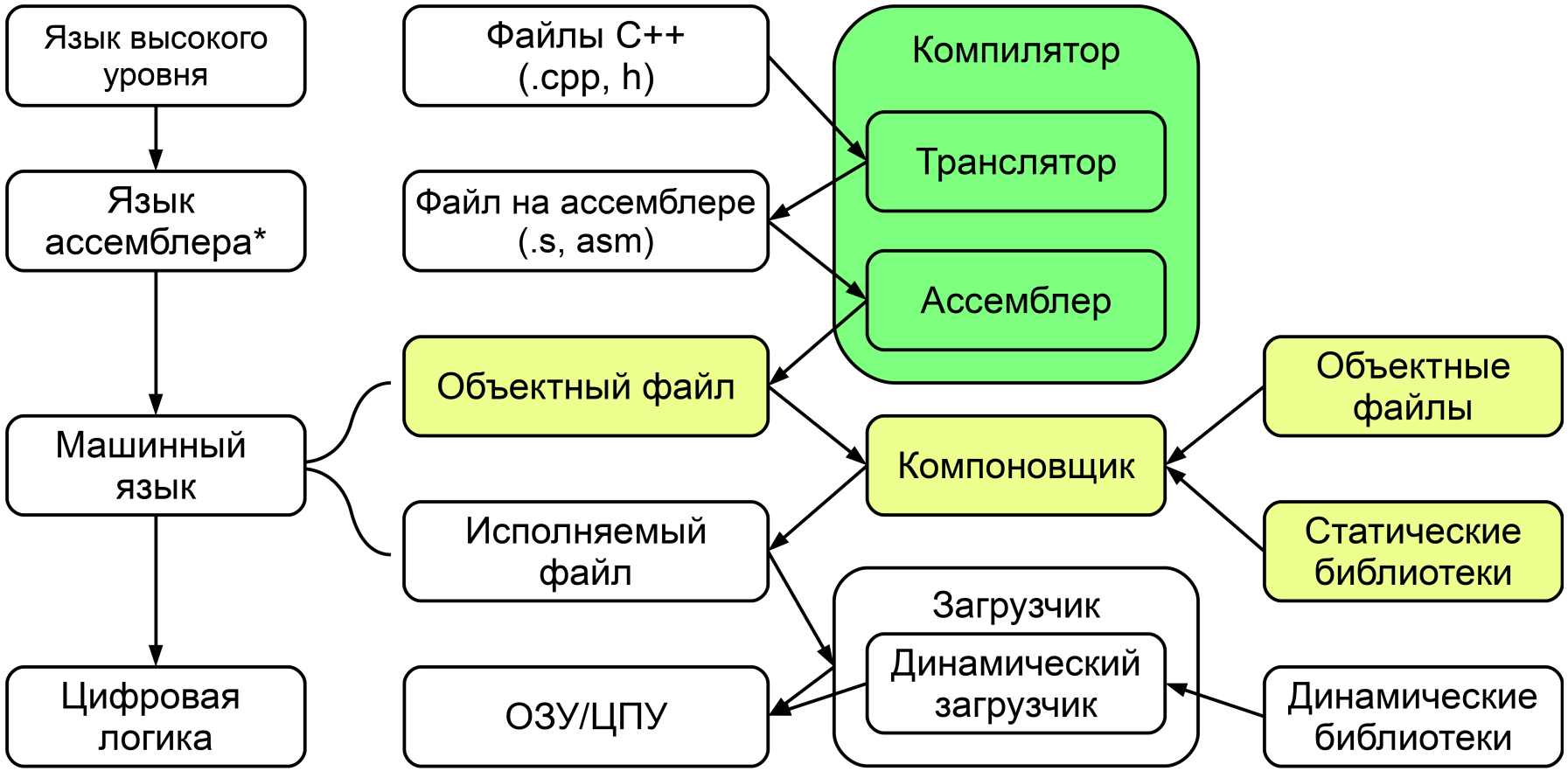
Компилятор связывает эти символические адреса с *перемещаемыми* адресами (такими как N байт от начала модуля).

Загрузчик или **компоновщик** (linker), в свою очередь, связывают эти перемещаемые адреса с *виртуальными/физическим* адресами и создают *исполняемый файл*.

Этап загрузки (Load time). Если на стадии компиляции не известно где процесс будет размещен в памяти, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.

Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одного сегмента памяти в другой, связывание откладывается до времени выполнения. Здесь желательно специализированное оборудование, например регистры перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Например, x86 использует четыре таких (сегментных) регистра.

Связь между файлами программы и утилитами

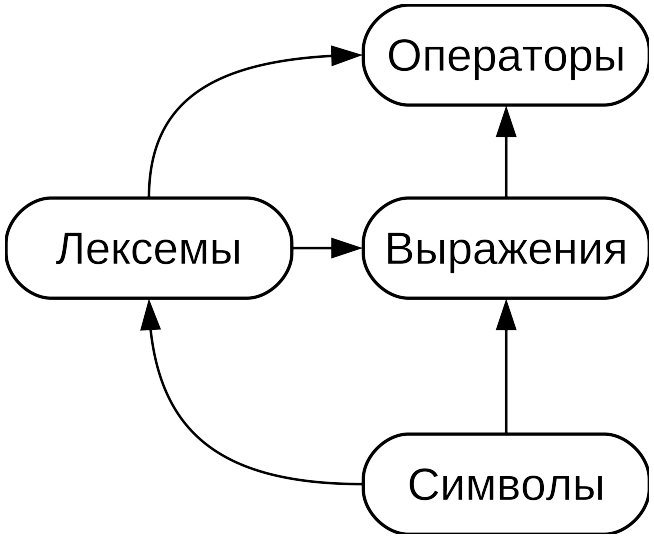


make — утилита, отслеживающая изменения в файлах и вызывающая необходимые программы из набора, использующегося для компиляции и генерации выполняемого кода из исходных текстов (toolchain).

Понятие алфавита

В общем случае язык состоит из знаковой системы — множества допустимых последовательностей знаков, множества смыслов этой системы, соответствия между последовательностями знаков и смыслами.

Естественный язык	Алгоритмический язык
Предложения Словосочетания Слова Символы	Операторы Выражения Лексемы Символы



Символ (буква) – это простой атомарный (неделимый) знак.
Лексема – элементарная конструкция, минимальная единица языка, имеющая самостоятельный смысл.
Выражение задает правила **вычисления некоторого значения**.
Оператор задает законченное описание **некоторого действия**.

Для описания сложных действий требуется последовательность операторов. Операторы могут объединяться в блоки.

Алфавит формального языка — множество атомарных (неделимых) символов какого-либо формального языка. [Алфавит A]. В математической логике и дискретной математике это синоним множества. В информатике — конечное множество символов или цифр.

Из символов алфавита формального языка строятся слова, а с помощью задания *формальной грамматики* — допустимые выражения языка.

Цепочки символов

Цепочка символов (строка) – это произвольная упорядоченная конечная последовательность символов некоторого алфавита.

Свойства:

Произвольность — в цепочку может входить любая допустимая последовательность символов, она не обязательно должна иметь смысл.

Упорядоченность — цепочка имеет определенный состав входящих в нее символов, их количество и порядок следования

Обычно цепочки обозначают буквами греческого алфавита $\alpha, \beta, \gamma \ldots$

1) ***Эквивалентность***: α = β, если цепочки α и β имеют один и тот же состав символов, их количество и порядок следования.

2) ***Длина цепочки*** – это количество символов в ней: |α|. Если α = β ⇒ |α| = |β|

3) ***Конкатенацией*** Θ цепочек α, β называется цепочка γ = α Θ β = αβ. |γ| = |α| + |β| .
Цепочка α является префиксом, β - суффиксом цепочки γ.
Свойства:
- некоммутативность: ∃ α, β: αβ ≠ βα
- ассоциативность: γ (αβ) = (γα) β

Любую цепочку символов можно представить как конкатенацию составляющих ее частей.

4) Цепочка ω называется ***подцепочкой*** γ, если γ = αωβ.

5) ***Заменой*** (***подстановкой***) цепочки γ = αωβ называется новая цепочка γ ’ = αφβ

6) ***Обращение*** цепочки α^R – это запись символов цепочки α в обратном порядке.

$\alpha^{RR} = \alpha$.

$\forall \alpha, \beta : (\alpha\beta)^R = \beta^R \alpha^R$

(похоже на сопряжение)

7) ***Итерация*** цепочки α^n — это конкатенация цепочки α самой с собой n раз. $n \in \mathbb{N}, n \geqslant 0$

$\alpha^n = \alpha\alpha^{n-1} = \alpha^{n-1}\alpha$

8) ***Пустая цепочка*** символов λ не содержит ни одного символа (иногда обозначают как ε).

$\alpha^0 = \lambda$

$|\lambda| = 0$

$\forall \alpha : \lambda\alpha = \alpha\lambda = \alpha$

$\lambda^R = \lambda$

$\forall n \geqslant 0 : \lambda^n = \lambda$

9) Цепочка символов α является цепочкой над алфавитом A : $\alpha(\mathsf{A})$, если $\forall x \in \alpha, x \in \mathsf{A}$

10) Замыкание алфавита A (A^*) — это множество всех возможных цепочек над алфавитом A

Примеры

$\mathsf{A} = \{a, b, c\}$

$\mathsf{A}^* = \{\lambda, a, b, c, aa, ab, c, bb, bc, ba, cc, ca, cb, aaa, aab, aac, \dots\}$

$\mathsf{A}* = \mathsf{A}^0 \cup \mathsf{A}^1 \cup \mathsf{A}^2 \cup \dots,$

где $\mathsf{A}^0 = \{\lambda\}, \mathsf{A}^1 = \mathsf{A}, \mathsf{A}^2 = \mathsf{A}^1 \otimes \mathsf{A}, \dots, \mathsf{A}^{k-1} \otimes \mathsf{A}, \dots$

$\mathsf{A}^+ = \mathsf{A}^1 \cup \mathsf{A}^2 \cup \dots, = \mathsf{A}^* \setminus \mathsf{A}^0$

$\mathsf{A}^* = \mathsf{A}^+ \cup \{\lambda\}$

Язык

1) Язык $\mathcal{L}$ над алфавитом A или $\mathcal{L}(\mathrm{A})$— это некоторое счетное подмножество цепочек конечной длины из множества всех цепочек алфавита A такое, что $\mathcal{L}(\mathrm{A}) \subseteq \mathrm{A}^*$.

2) Язык $\mathcal{L}(\mathrm{A})$ включает в себя язык $\mathcal{L}'(\mathrm{A})$ или $\mathcal{L}'(\mathrm{A}) \subseteq \mathcal{L}(\mathrm{A})$, если для $\forall \alpha \in \mathcal{L}'(\mathrm{A})$ справедливо $\alpha \in \mathcal{L}(\mathrm{A})$.

Т.е. множество цепочек языка $\mathcal{L}'(\mathrm{A})$ является подмножеством множества цепочек языка $L(A)$.

3) Языки $\mathcal{L}(\mathrm{A})$ и $\mathcal{L}'(\mathrm{A})$ эквивалентны, или $\mathcal{L}'(\mathrm{A}) = \mathcal{L}(\mathrm{A})$, если $\mathcal{L}'(\mathrm{A}) \subseteq \mathcal{L}(\mathrm{A})$ и $\mathcal{L}(\mathrm{A}) \subseteq \mathcal{L}'(\mathrm{A})$.

Для эквивалентных языков множества допустимых цепочек равны

Языки $\mathcal{L}(\mathrm{A})$ и $\mathcal{L}'(\mathrm{A})$ почти эквивалентны:

$\mathcal{L}'(\mathrm{A}) \cong \mathcal{L}(\mathrm{A})$, если $\mathcal{L}'(\mathrm{A}) \cup \{\lambda\}$ = $\mathcal{L}(\mathrm{A}) \cup \{\lambda\}$.

4) $\mathcal{L}\{\lambda\} = \{\lambda\}\mathcal{L} = \mathcal{L}; \quad \mathcal{L}\varnothing = \varnothing\mathcal{L} = \varnothing$.

5) Конкатенация произвольного числа цепочек формального языка $\mathcal{L}$ носит название ***замыкания Клини***:

$\mathcal{L}^* = \mathcal{L}^0 \cup \mathcal{L}^1 \cup \mathcal{L}^2 \cup \ldots$, где $\mathcal{L}^0 = \{\lambda\}$.

При $n \geqslant 1$ справедливо $\mathcal{L}^n = \mathcal{L}\mathcal{L}^{n-1} = \mathcal{L}^{n-1}\mathcal{L}$ — ноль или более сцеплений языка $\mathcal{L}$:

Позитивное замыкание означает одно или более сцеплений языка $\mathcal{L}$: $\mathcal{L}^+ = \mathcal{L}^1 \cup \mathcal{L}^2 \cup \ldots$

или $\mathcal{L}^+ = \mathcal{L}\mathcal{L}^* = \mathcal{L}^*\mathcal{L}$, $\mathcal{L}^* = \mathcal{L}^+ \cup \{\lambda\}$.

Способы заданий языков

- перечисление всех допустимых цепочек языка;
- указание способа порождения цепочек языка (определение грамматики);
- указание метода распознавания цепочек языка (определение распознавателя).

Лексика, синтаксис, семантика языка

Лексика – это совокупность слов (словарный запас) языка;

Лексема (слово, лексическая единица) языка — это конструкция, которая состоит из элементов алфавита языка и не содержит в себе других конструкций;

Синтаксис – набор правил, определяющий допустимые конструкции языка, т.е. определение набора цепочек символов, принадлежащих языку.

В виде строгого набора правил можно описать только формальные языки.

Для большинства ЯП набор заданных синтаксических правил нуждается в дополнительных пояснениях.

Математический аппарат для изучения синтаксиса языков называется *теорией формальных грамматик*.

Семантика – это раздел языка, определяющий значение предложений языка (смысл всех допустимых цепочек языка).

ЯП занимают промежуточное положение между формальными и естественными языками. Как и формальные языки, они имеют строгие синтаксические правила. Из естественных языков ЯП позаимствовали ряд слов, выражающих ключевые слова. Для задания ЯП необходимо:

- определить множество допустимых символов языка;
- определить множество правильных программ языка;
- задать смысл для каждой правильной программы.

Формальные грамматики

Ранее было определение языка $\mathcal{L}$ над алфавитом A, как подмножества цепочек A^*.

Это очень общее определение, не позволяющее выделять среди множества языков отдельные их классы.

Соотношения Туэ – это правила, согласно которым любой цепочке $\alpha = \gamma\xi\delta$ из множества A^* ставится в соответствие цепочка $\beta = \gamma\eta\delta$ из того же множества.

Ограничения на них в виде введения односторонних правил привели к созданию формального математического аппарата – к ***формальным грамматикам***.

Теория формальных грамматик занимается описанием, распознаванием и переработкой языков.

Решение прикладных вопросов:
- могут ли языки из некоторого класса Z быть легко распознанными;
- принадлежит ли данный язык классу Z;
- существуют ли алгоритмы, определяющие принадлежность цепочки α языку $\mathcal{L}$ и т.д.

Способы описания отдельных классов языков:
- с помощью порождающей процедуры (задается конечным множеством правил, называемых ***грамматикой***);
- с помощью распознающей процедуры (задается абстрактным распознающим устройством – ***автоматом***).

- ***Правило*** (***продукция***) – упорядоченная пара цепочек символов (α, β).
- $\alpha \to \beta$ (α порождает β)
- Язык, заданный грамматикой G: $\mathcal{L}(\mathsf{G})$
- Грамматики G и G′ называются эквивалентными, если определяют один и тот же язык:

$$\mathcal{L}(\mathsf{G}) = \mathcal{L}(\mathsf{G}')$$

- Грамматики почти эквивалентны:

$$\mathcal{L}(\mathsf{G}) \cup \{\lambda\} = \mathcal{L}(\mathsf{G}') \cup \{\lambda\}$$

Формально грамматика определяется как четверка $\mathsf{G}(\mathsf{T}, \mathsf{N}, \mathsf{P}, \mathsf{S})$

T - конечное непустое множество терминальных символов языка (терминальный или основной словарь грамматики G, строчные символы a,b,c....);

N – конечное непустое множество нетерминальных символов (нетерминальный или вспомогательный словарь, прописные символы A,B,C,...);

$$\mathsf{N} \cap \mathsf{T} = \varnothing$$

Множество $\mathsf{V} = \mathsf{N} \cup \mathsf{T}$ — полный алфавит (объединенный словарь) грамматики G (vocabulary — словарь, лексика, лексикон, терминология, запас слов, словарный состав).

P – конечное множество правил (продукций) грамматики вида $\alpha \to \beta$,
где $\alpha \in (\mathsf{N} \cup \mathsf{T})^+$, $\in (\mathsf{N} \cup \mathsf{T})^+$
S – начальный символ (аксиома) грамматики, $\mathsf{S} \in \mathsf{N}$.
S обозначает главный нетерминал, цель грамматики G.

Цепочка ω' ***непосредственно выводима*** из цепочки ω в грамматике G ($\omega \Rightarrow \omega'$), если

$$\omega = \xi_1 \varphi \xi_2, \omega' =_1 \psi \xi_2 \text{ и } \exists\, (\varphi \to \psi) \in \mathsf{P}$$

Цепочка ω' ***выводима*** из цепочки ω в грамматике G ($\omega \Rightarrow^* \omega'$), если $\exists$ последовательность цепочек

$$\omega = \omega_0, \omega_1, \ldots, \omega_n = \omega' : \omega_k \Rightarrow \omega_{k+1},\ k = 0, 1, \ldots, n-1, \text{ либо } \omega = \omega'.$$

Последовательность цепочек $\omega = \omega_0, \omega_1, \ldots, \omega_n$ называется ***выводом*** цепочки ω_n из цепочки ω_0 в грамматике G.

ω_n может быть выведена как за ноль шагов ($\omega = \omega'$), так и за один и более шагов ($\omega \Rightarrow^+ \omega'$).

Каждая строка, которую можно вывести из аксиомы – ***сентенциальная форма***. Если она состоит только из терминалов, то представляет собой строку языка.

Язык $\mathcal{L}$, порождаемый грамматикой G – это множество всех цепочек терминальных символов, выводимых из аксиомы грамматики:

$$\mathcal{L}(\mathsf{G}) = \{\chi \mid S \Rightarrow^* \chi,\ \chi \in T^*\}$$

Пример грамматики реляционного выражения :

$\mathsf{G} = \langle \mathsf{T}, \mathsf{N}, \mathsf{P}, \mathsf{S} \rangle$

N = {F, D, C}, T = {>, <, =, !}, S = C

P = {C → D=, C → F, D → !, D → =, D → F, F → <, F → >}

P = {C → D= | F, D → ! | = | F, F → < | >}

C ⇒ D= ⇒ F= ⇒ <=

- грамматике не присуща детерминированность – конкретный порядок подстановки правил (алгоритм) является произвольным, не определенным строго;

- компактность;

- алгоритм можно зафиксировать различными способами, поэтому формальная грамматика потенциально задает множество алгоритмов порождения языка.

Форма Бэкуса-Наура (BNF)

Явилась исторически первой для записи в сжатом виде привил грамматики:

- основной словарь – основные символы
- вспомогательный словарь – металингвистические переменные
- в левой части формулы (формы) находится металингвистическая переменная, обозначающая соответствующую конструкцию
- в правой части указываются способы построения конструкции
- варианты в правой части разделяются символом **|** , обозначающем ИЛИ
- правая и левая части разделяются связкой **::=**, что означает «определяется как»
- имена металингвистических переменных заключаются в угловые скобки **< >**

Пример для вышеописанной грамматики:

```
<compare> ::= <double>= | <first>
<double>  ::= ! | = | <first>
<first>   ::= < | >
```

Рекурсия

Возможность описания бесконечного множества цепочек языка с помощью конечного набора правил достигается за счет ***рекурсии:***

Явная – символ определяется сам через себя A → A

Неявная (косвенная) – символ определяется через цепочку правил: A → Dx, D → A

Чтобы рекурсия не была бесконечной, должны существовать и другие правила, определяющие тот же символ: A → A | Dx

ФБН не позволяют описывать контекстные зависимости ЯП, например, запрет на повторное описание одного и того же идентификатора в пределах блока.

Для этого используются другие, метасемантические языки. ФБН, как правило, является их ядром.

Правила грамматики могут быть записаны и в другой форме. Например, для описания синтаксиса КОБОЛа и ПЛ/1 использовалось расширение ФБН.

Синтаксические диаграммы

Еще один способ описания – ***синтаксические диаграммы***.

Впервые была использована при описании языка Pascal.

Эта форма записи доступна для тех грамматик, в которых в левой части присутствует не более одного символа. Этого достаточно для описания всех существующих ЯП. В данной форме каждому нетерминалу соответствует диаграмма в виде направленного графа. Типы вершин:

точка входа – на диаграмме не обозначается, из нее просто начинается входная дуга графа;

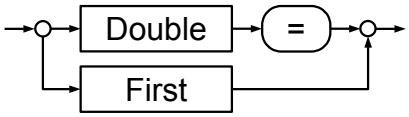
нетерминал – на диаграмме обозначается прямоугольником, в котором написано его обозначение;

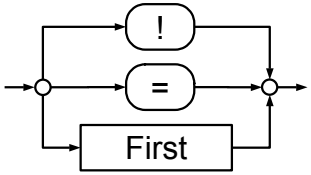
цепочка терминалов – обозначается овалом, внутри которого записана цепочка;

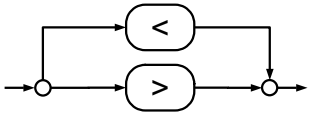
узловая точка – обозначается точкой или закрашенным кружком;

точка выхода – никак не обозначается, в нее просто входит выходная дуга графа;

Каждая диаграмма имеет только одну точку входа и одну точку выхода. Вершин остальных типов может быть сколько угодно. Вершины соединяются направленными дугами. Из входной точки дуги только выходят, в выходную – только входят. Остальные вершины должны иметь как минимум один вход и один выход.







Распознаватели