Классика программирования

Никлаус Вирт

Построение компиляторов

CD

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

УДК 32.973.26-018.2
ББК 004.438
В52

Никлаус Вирт
Построение компиляторов / Пер. с англ. Борисов Е. В., Чернышов Л. Н. – М.: ДМК Пресс, 2013. – 192 с.: ил.

ISBN 978-5-94074-875-5

Книга известного специалиста в области информатики Никлауса Вирта написана по материалам его лекций по вводному курсу проектирования компиляторов. На примере простого языка Оберон-0 рассмотрены все элементы транслятора, включая оптимизацию и генерацию кода. Приведен полный текст компилятора на языке программирования Оберон.

Для программистов, преподавателей и студентов, изучающих системное программирование и методы трансляции.

Содержание компакт-диска:

Базовая конфигурация системы Блэкбокс с коллекцией модулей, реализующих оригинальный компилятор с языка Оберон-0 и компилятор, адаптированный под Блэкбокс.
Базовые инструкции по работе в системе Блэкбокс.
Полный перевод документации системы Блэкбокс на русский язык.
Конфигурация системы Блэкбокс для использования во вводных курсах программирования в университетах.
Конфигурация системы Блэкбокс для использования в школах (полная русификация меню, сообщений компилятора, с возможностью использования ключевых слов на русском и других национальных языках).
Доклады участников проекта Информатика-21 по опыту использования системы Блэкбокс в обучении программированию.
Оригинальные дистрибутивы системы Блэкбокс 1.5 (основной рабочий) и 1.6rc6.
Инструкции по работе в Блэкбоксе под Linux/Wine.
Дистрибутив оптимизирующего компилятора XDS Oberon (версии Linux и MS Windows).
OberonScript – аналог JavaScript для использования в Web-приложениях.

This is a slightly revised version of the book published by Addison-Wesley in 1996

ISBN 0-201-40353-6 (анг.)
ISBN 978-5-94074-875-5

Содержание

Глава 2

Язык и синтаксис

Каждый язык обладает структурой, называемой грамматикой, или синтаксисом. Например, правильное предложение (в английском языке – *Прим. перев.*) всегда состоит из подлежащего и следующего за ним сказуемого. Под правильным здесь понимается *правильно составленное* предложение. Это можно описать следующей формулой:

```
предложение = подлежащее сказуемое.
```

Если мы добавим к этой формуле еще две

```
подлежащее  = "Джон" | "Мария".
сказуемое   = "ест" | "говорит".
```

то с их помощью получим ровно четыре возможных предложения, а именно:

```
Джон ест          Мария ест
Джон говорит      Мария говорит
```

где символ | должен произноситься как *или*. Мы назовем эти формулы *синтаксическими правилами*, *продукциями* или просто *синтаксическими уравнениями*. Подлежащее и сказуемое – это синтаксические классы. Краткая запись этих формул пренебрегает смыслом идентификаторов:

```
S = AB.           L = {ac, ad, bc, bd}
A = "a" | "b".
B = "c" | "d".
```

Мы будем использовать такую сокращенную запись в последующих кратких примерах. Множество L предложений, которые могут быть сгенерированы этим способом, то есть повторяющейся заменой левых частей уравнений правыми, называется *языком*.

Приведенный выше пример, очевидно, определяет язык, состоящий только из четырех предложений. Обычно язык содержит бесконечно много предложений. Следующий пример показывает, что бесконечное множество может быть очень просто определено конечным числом уравнений. Символ ∅ обозначает пустую последовательность.

```
S = A.            L = {∅, a, aa, aaa, aaaa,...}
A = "a" A | ∅.
```

Метод, позволяющий выполнять подстановку (здесь "a"A вместо A) бесконечное число раз, называется *рекурсией*.

Наш третий пример опять основан на применении рекурсии. Но он генерирует не только предложения, состоящие из произвольной последовательности одного и того же символа, но и вложенные предложения:

```
S = A.            L = {b, abc, aabcc, aaabccc,...}
A = "a" A "c" | "b".
```

Понятно, что таким образом может быть выражена произвольно глубокая вложенность (здесь – для A), что особенно важно в определении структурированных языков.

Наш четвертый, и последний, пример показывает структуру выражений. Символы E, T, F и V обозначают выражение, слагаемое, множитель и переменную соответственно.

```
E = T | "+" T.
T = F | T "*" F.
F = V | "(" E ")".
V = "a" | "b" | "c" | "d".
```

Из этого примера видно, что синтаксис не только определяет множество предложений языка, но и наделяет их структурой. Синтаксис раскладывает предложения на составляющие, как показано в примере на рис. 2.1. Графические представления называются *структурными деревьями*, или *синтаксическими деревьями*.

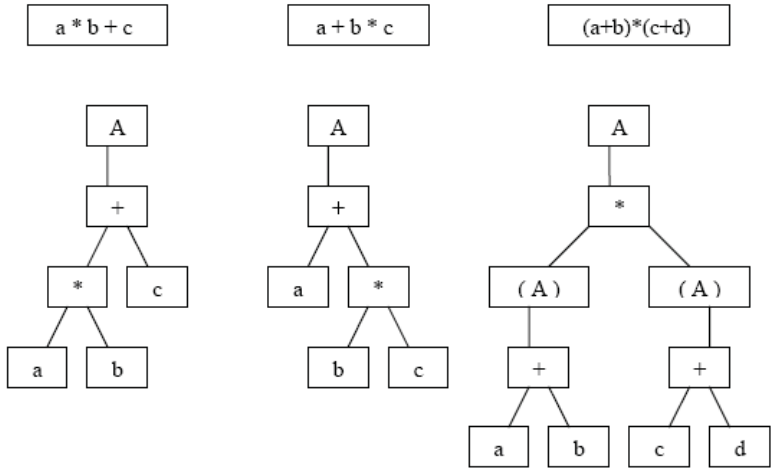

Рис. 2.1. Структура выражений

Сформулируем представленные выше понятия более строго.

Язык (порождающая язык грамматика – *Прим. перев.*) определяется следующим образом:

1. Множество *терминальных символов*. Это символы, которые появляются в предложениях. Говорят, что они терминальные, потому что не могут быть заменены никакими другими символами. Процесс подстановки заканчивается терминальными символами. В нашем первом примере это множество состоит из элементов a, b, c и d. Это множество также называется *словарем*.
2. Множество *нетерминальных символов*. Они обозначают синтаксические классы и могут замещаться в результате подстановок. В нашем первом примере это множество состоит из элементов S, A и B.
3. Множество *синтаксических уравнений* (также называемых *продукциями*). Они определяют возможные подстановки нетерминальных символов. Уравнение задается для каждого нетерминального символа.

4. *Начальный символ*. Это нетерминальный символ, обозначаемый в примерах как S.

Таким образом, язык – это множество цепочек терминальных символов, которые могут быть выведены из начального символа многократным применением синтаксических уравнений, то есть подстановок.

Желательно также строго и точно определить нотацию, в которой записываются синтаксические уравнения. Пусть нетерминальные символы будут идентификаторами, как в языках программирования, то есть последовательностями букв (и, возможно, цифр), например expression, term. Пусть терминальные символы будут последовательностями символов, заключенными в кавычки (строками), например "=", "|". Для определения структуры этих уравнений удобно воспользоваться тем же самым инструментом, который только что был определен:

```
syntax      = production syntax | ∅.
production  = identifier "=" expression ".".
expression  = term | expression "|" term.
term        = factor | term factor.
factor      = identifier | string.
identifier  = letter | identifier letter | identifier digit.
string      = stringhead"'".
stringhead  = "'" | stringhead character.
letter      = "A" | ... | "Z".
digit       = "0" | ... | "9".
```

Эта нотация почти в таком же виде была введена в 1960 году Дж. Бэкусом и П. Науром для формального описания синтаксиса языка Алгол 60 и поэтому получила название формы Бэкуса–Наура (БНФ) [12]. Как показывает пример, использование рекурсии для простых повторений несколько мешает их восприятию. Поэтому мы расширим эту нотацию двумя конструкциями, выражающими повторение и необязательность. Кроме этого, разрешим выражения заключать в скобки. Таким образом, вводится расширение БНФ, называемое РБНФ [17], которым мы снова воспользуемся для его же точного определения:

```
syntax      = { production }.
production  = identifier "=" expression ".".
expression  = term {"|" term }.
term        = factor { factor }.
factor      = identifier | строка | "(" expression ")" | "[" expression "]"
              | "{" expression "}".
identifier  = letter { letter | digit }.
string      = "'" {character} "'".
letter      = "A" |... | "Z".
digit       = "0" |... | "9".
```

Множитель вида {x} равнозначен произвольно длинной последовательности x, включая пустую последовательность. Продукция вида

```
A = AB | ∅.
```

теперь записывается короче: A = {B}. Множитель вида [x] равнозначен «x, или ничто», то есть выражает необязательность. Следовательно, потребность в специальном символе ∅ для пустой цепочки исчезает.

Идея определять языки и их грамматику с математической точностью восходит к Н. Хомскому (N. Chomsky). Однако стало ясно, что предложенная простая схема правил подстановки недостаточна для представления всей сложности разговорных языков. Положение не изменилось даже после того, как формализм был значительно расширен. Но зато эта работа оказалась чрезвычайно плодотворной для теории языков программирования и математических формализмов. С его помощью Алгол 60 стал первым языком программирования, который был определен точно и формально. Мимоходом подчеркнем, что эта точность относилась только к синтаксису, но не к семантике.

Термин *язык программирования* также обязан формализму Хомского, поскольку языки программирования, оказывается, обладают структурой, подобной структуре разговорных языков. Но мы уверены, что этот термин, в общем, довольно неудачен, поскольку на языке программирования нельзя разговаривать, и поэтому он не язык в прямом смысле слова. Более подходящими были бы термины формализм или формальная нотация.

Некоторые удивляются, почему точному определению предложений языка должно придаваться такое большое значение; ведь в действительности это не так. Тем не менее очень важно, чтобы предложение было правильно составлено. Хотя и в этом случае предложение может потребовать уточнения. Но, в конце концов, структура предложения (правильно составленного) важна потому, что является инструментом понимания его смысла. Благодаря синтаксической структуре отдельные составные части предложения и их смысл могут распознаваться независимо, а все вместе они придают смысл целому.

Давайте проиллюстрируем этот момент, используя следующий простой пример выражения со сложением. Обозначим идентификатором E выражение, а N – число:

```
E = N | E "+" E.
N = "1" | "2" | "3" | "4".
```

Очевидно, "4 + 2 + 1" – правильное выражение, которое можно получить несколькими способами, каждому из которых соответствует своя структура, как показано на рис. 2.2.

Обе структуры могут быть представлены скобочными выражениями, а именно (4 + 2) + 1 и 4 + (2 + 1) соответственно. К счастью, благодаря ассоциативности сложения результат в обоих случаях один и тот же и равен 7. Однако это не всегда так. Если в нашем примере знак сложения заменить на знак вычитания, то две структуры дадут разные результаты: (4 – 2) – 1 = 1, 4 – (2 – 1) = 3.

Пример иллюстрирует два факта:

1. Интерпретация предложений всегда основывается на распознавании синтаксической структуры.
2. Каждое предложение должно иметь единственную структуру, для того чтобы не быть двусмысленным.

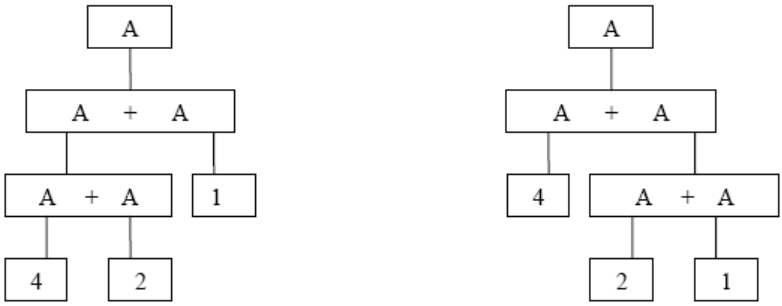

Рис. 2.2. Разные структурные деревья для одного и того же выражения

Если второе требование не выполняется, может возникнуть двусмысленное предложение. Этим можно обогатить разговорный язык; однако двусмысленные языки программирования просто бесполезны.

Мы называем синтаксический класс неоднозначным, если ему можно соотнести несколько структур. Язык неоднозначен, если в него входит по крайней мере один неоднозначный синтаксический класс (конструкция).

2.1. Упражнения

2.1. Сообщение об Алголе 60 содержит следующий синтаксис (приведенный к РБНФ):

```
primary   = unsignedNumber | variable | "(" arithmeticExpression ")" |... .
factor    = primary | factor "↑" primary.
term      = factor | term ("*" | "/" | "÷") factor.
simpleArithmeticExpression = term | ("+" | "–") term
                                  | simpleArithmeticExpression ("+" | "–") term.
arithmeticExpression = simpleArithmeticExpression |
                       "IF" BooleanExpression "THEN" simpleArithmeticExpression
                       "ELSE" arithmeticExpression.
relationalOperator = "=" | "≠"| "≤" | "<" | "≥" | ">".
relation           = arithmeticExpression relationalOperator arithmeticExpression.
BooleanPrimary = logicalValue | variable | relation
                 | "("BooleanExpression")" |... .
BooleanSecondary  = BooleanPrimary | "¬" BooleanPrimary.
BooleanFactor     = BooleanSecondary | BooleanFactor "∩" BooleanSecondary.
BooleanTerm       = BooleanFactor | BooleanTerm "⊆" BooleanFactor.
implication       = BooleanTerm | implication "=>" BooleanTerm.
simpleBoolean     = implication | simpleBoolean "≡" implication.
BooleanExpression = simpleBoolean |
          "IF" BooleanExpression "THEN" simpleBoolean "ELSE" BooleanExpression.
```

Определите синтаксические деревья следующих выражений, в которых буквы обозначают переменные:

```
x + y + z
x * y + z
x + y * z
(x – y) * (x + y)
–x ÷ y
a + b < c + d
a + b < c V d ≠ e ∩ ¬f => g>h ? i *j = k ↑ L V m–n + p ≤ q
```

2.2. Следующие продукции также являются частью первоначального определения Алгола 60. Они содержат двусмысленности, которые были устранены в Пересмотренном сообщении.

```
forListElement = arithmeticExpression
 | arithmeticExpression "STEP" arithmeticExpression "UNTIL" arithmeticExpression
 | arithmeticExpression "WHILE" BooleanExpression .
forList            = forListElement | forList "," forListElement .
forClause          = "FOR" variable ":=" forList "DO" .
forStatement       = forClause statement .
compoundTail       = statement "END" | statement ";" compoundTail .
compoundStatement  = "BEGIN" compoundTail .
unconditionalStatement = basicStatement | forStatement | compoundStatement|...
ifStatement                = "IF" BooleanExpression "THEN" unconditionalStatement.
conditionalStatement  = ifStatement | ifStatement "ELSE" statement .
statement                  = unconditionalStatement | conditionalStatement .
```

Найдите по крайней мере две различные структуры для следующих выражений и операторов. Пусть A и B обозначают «простые операторы».

```
IF a THEN b ELSE c = d
IF a THEN IF b THEN A ELSE B
IF  a  THEN  FOR ... DO  IF  b  THEN  A  ELSE  B
```

Предложите альтернативный однозначный синтаксис.

2.3. Рассмотрите следующие конструкции и попытайтесь разобраться, какие из них являются корректными для Алгола, а какие – для Оберона (см. приложение 2):

```
a+b=c+d
a * –b
a<b  &  c<d
```

Вычислите следующие выражения:

```
5 * 13 DIV 4 =
13  DIV  5 *4 =
```

Глава 3

Регулярные языки

Синтаксические уравнения в том виде, как они определены в РБНФ, генерируют контекстно-свободные языки. Термин «контекстно-свободный» принадлежит Хомскому и происходит из того факта, что замена символа слева от знака = цепочкой, порожденной выражением справа от знака =, возможна всегда, независимо от контекста этого символа внутри предложения. Оказывается, что эта свобода контекста (по Хомскому) очень подходит и даже желательна для языков программирования. Но контекстная зависимость в *ином* смысле все-таки необходима. Мы вернемся к этой теме в главе 8.

Здесь мы прежде всего хотим исследовать подкласс, а не контекстно-свободные языки вообще. Этот подкласс, известный как *регулярные* языки, играет существенную роль в области языков программирования. В сущности, это контекстно-свободные языки, синтаксис которых не содержит иной рекурсии, кроме спецификации повторения. Так как в РБНФ повторение определено непосредственно и без использования рекурсии, можно дать простое определение:

> Язык *регулярен*, если его синтаксис может быть выражен единственным выражением РБНФ.

Требование достаточности единственного уравнения подразумевает, что выражение состоит только из терминальных символов. Такое выражение называют регулярным выражением.

Видимо, достаточно двух кратких примеров регулярных языков. Первый определяет идентификаторы, поскольку они встречаются в большинстве языков, а второй – целые числа в десятичной записи. Для краткости мы используем нетерминальные символы letter и digit. Их можно исключить подстановкой, в результате чего и identifier, и integer будут регулярными выражениями.

```
identifier = letter {letter | digit}.
integer = digit {digit}.
letter = "A" | "B" | ... | "Z"
digit = "0" |"1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Причина нашего интереса к регулярным языкам заключается в том, что программы для распознавания регулярных выражений очень просты и эффективны. Под «распознаванием» мы подразумеваем определение структуры предложения и, следовательно, его правильность, то есть принадлежность языку. Распознавание выражения называется *синтаксическим анализом*.

Для распознавания регулярных выражений необходимо и достаточно иметь конечный автомат, также называемый *машиной состояний* (*state machine*). На каждом шаге машина состояний читает следующий символ и меняет свое состояние. Следующее состояние определяется исключительно предыдущим состоянием и очередным прочитанным символом. Если очередное состояние уникально, то машина состояний является *детерминированной*, иначе *недетерминированной*. Если машина состояний реализована программно, то состояние машины определяется текущей точкой выполнения программы.

Анализирующая программа может быть выведена непосредственно из определения синтаксиса в РБНФ. Для каждой РБНФ-конструкции **K** существует прави-

ло перевода, которое порождает фрагмент программы **Pr(K)**. Правила перевода из РБНФ в текст программы показаны ниже. В них sym – глобальная переменная, представляющая собой последний символ исходного текста, прочитанный процедурой next. Процедура error завершает выполнение программы, сигнализируя о том, что последовательность символов не принадлежит языку.

K	**Pr(K)**
"x"	IF sym = "x" THEN next ELSE error END
(exp)	Pr(exp)
[exp]	IF sym IN first(exp) THEN Pr(exp) END
{exp}	WHILE sym IN first(exp) DO Pr(exp) END
fac0 fac1 ... facN	Pr(fac0); Pr(fac1); ... Pr(facN);
term0|term1|...|termN	CASE sym OF first(term0): Pr(term0); | first(term1): Pr(term1); ... | first(termN): Pr(termN); END

Множество first(**K**) содержит все символы, с которых могут начинаться предложения, полученные из конструкции **K**. Это множество *начальных символов* конструкции **K**. Для двух примеров – идентификаторов и целых чисел – они таковы:

```
first(integer) = digits = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"}
first(identifier) = letters = {"A", "B",..., "Z"}
```

Однако применение этих простых правил перевода, генерирующих синтаксический анализатор по заданному синтаксису, возможно лишь при условии детерминированного синтаксиса. Это предварительное условие может быть сформулировано более конкретно следующим образом:

K	**Cond(K)**
term0 | terml	Символы term0 и term1 не должны иметь никаких общих начальных символов
fac0 fac1	Если fac0 содержит пустую последовательность, то fac0 и fac1 не должны иметь никаких общих начальных символов
[exp] или {exp}	Множества начальных символов exp и символов, которые могут следовать за **K**, не должны пересекаться

Очевидно, эти условия выполняются в примерах для идентификаторов и целых чисел, и мы, таким образом, получаем следующие программы для их распознавания:

```
IF s IN letters THEN next ELSE error END;
WHILE sym IN letters + digits DO
    CASE sym OF
      "A" .. "Z": next
    | "0" .. "9": next
    END
END
```

```
IF sym IN digits THEN next ELSE error END;
WHILE sym IN digits DO next END
```

Обычно программу, полученную применением правил перевода, можно упростить путем устранения условий, которые уже явно установлены предыдущими условиями. Условия sym IN letters и sym IN digits записываются следующим образом:

```
("A" <= sym) & (sym <= "Z")
("0" <= sym) & (sym <= "9")
```

Важность регулярных языков для языков программирования следует из того, что последние обычно определяются в два этапа. На первом этапе их синтаксис определяется в терминах словаря *абстрактных* терминальных символов. На втором эти абстрактные символы определяются как цепочки *конкретных* терминальных символов, таких как ASCII-символы. Это второе определение обычно имеет регулярный синтаксис. Разделение на два этапа обладает тем преимуществом, что определение абстрактных символов и, таким образом, языка не зависит от конкретного представления, то есть от некоторого конкретного набора символов, используемого на некотором конкретном оборудовании.

Такое разделение оказывает влияние и на структуру компилятора. Процесс синтаксического анализа использует процедуру получения следующего абстрактного символа, а эта процедура, в свою очередь, распознает конкретные символы – цепочки из одной или более литер. Эту последнюю процедуру называют лексическим анализатором, а синтаксический анализ на этом втором, более низком уровне – *лексическим анализом*. Определение символов, выражаемых литерами, обычно задается в форме регулярного языка, и поэтому лексический анализатор – типичная машина состояний.

Суммируем различия между этими двумя уровнями следующим образом:

Процесс	Входной элемент	Алгоритм	Синтаксис
Лексический анализ	Литера	Лексический анализатор	Регулярный
Синтаксический анализ	Символ	Синтаксический анализатор	Контекстно-свободный

В качестве примера приведем лексический анализатор для разбора РБНФ. Его терминальные символы и их определение в терминах литер таковы:

```
symbol    = {blank} (identifier | string | "(" | ")" | "[" | "]"
                   | "{" | "}" | "|" | "=" | ".") .
identifier = letter {letter | digit}.
string    = """ {character} """.
```

Из этих определений мы выведем процедуру *GetSym*, которая при каждом вызове присваивает глобальной переменной *sym* числовое значение, представляющее следующий прочитанный символ. Если символ является идентификатором (identifier) или строкой (string), то конкретная цепочка литер присваивается дополнительной глобальной переменной *id*. Также отметим, что лексический анали-

затор обычно подразумевает следующее правило о пробелах и концах строк: пробелы и концы строк разделяют цепочки литер и другого назначения не имеют. Процедура *GetSym* на Обероне использует следующие объявления:

```
CONST IdLen = 32;
    ident = 0; literal = 2; lparen = 3; lbrak = 4; lbrace = 5; bar = 6;
    eql = 7; rparen = 8; rbrak = 9; rbrace= 10; period = 11; other = 12;

TYPE Identifier = ARRAY IdLen OF CHAR;

VAR ch: CHAR;
    sym: INTEGER;
    id: Identifier;
    R: Texts.Reader;
```

Отметим, что абстрактная операция чтения теперь представлена конкретным вызовом *Texts.Read(R,ch)*. R – это глобально объявленный объект типа *Reader*, являющийся источником исходного текста. Отметим также, что переменная *ch* должна быть глобальной, потому что по завершении *GetSym* она может содержать первую литеру следующего символа. Это нужно иметь в виду при очередном обращении к *GetSym*.

```
PROCEDURE GetSym;
    VAR i: INTEGER;
BEGIN (*пропуск пробелов*)
    WHILE ~R.eot & (ch <= " ") DO Texts.Read(R, ch) END ;
    CASE ch OF
    "A".. "Z", "a" .. "z":
        sym := ident; i := 0;
        REPEAT id[i] := ch; INC(i);
                Texts.Read(R, ch)
        UNTIL (CAP(ch) < "A") OR (CAP(ch) > "Z");
        id[i] := 0X
    | 22X: (*кавычка*)
        Texts.Read(R, ch); sym := literal; i := 0;
        WHILE (ch # 22X) & (ch > "") DO
            id[i] := ch; INC(i); Texts.Read(R, ch)
        END;
        IF ch <= " " THEN error(l) END ;
        id[i] := 0X; Texts.Read(R, ch)
    | "=": sym := eql; Texts.ReadfR, ch)
    | "(" : sym := lparen; Texts.Read(R, ch)
    | ")" : sym := rparen; Texts.Read(R, ch)
    | "[" : sym := lbrak; Texts.Read(R, ch)
    | "]" : sym := rbrak; Texts.Read(R, ch)
    | "{" : sym := lbrace; Texts.Read(R, ch)
    | "}" : sym := rbrace; Texts.Read(R, ch)
    | "|" : sym := bar; Texts.Read(R, ch)
    | ".": sym := period; Texts.Read(R, ch)
```

```
    ELSE sym := other; Texts.Read(R, ch)
    END
END GetSym
```

3.1. Упражнение

Предложения регулярных языков могут быть распознаны конечными автоматами. Они обычно описываются диаграммами переходов. Каждый узел представляет состояние, а каждая стрелка – переход из одного состояния в другое. Стрелка помечена символом, который прочитан непосредственно перед переходом. Рассмотрите следующие диаграммы и опишите синтаксис соответствующих языков в РБНФ.