Федеральное агентство по образованию

АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Серия «Учебно-методический комплекс дисциплины»

Т.А. Галаган

Системное программное обеспечение

Учебное пособие

Благовещенск
2009

ББК
Г

Печатается по решению редакционно-издательского совета факультета математики и информатики Амурского государственного университета

Т.А. Галаган (составитель)

Системное программное обеспечение. Для студентов специальности 230102 – «Автоматизированные системы обработки информации и управления» очной формы обучения. – Благовещенск: Амурский гос. ун-т, 2009.

Пособие посвящено теоретическим основам разработки современного программного обеспечения. Рассмотрены теория формальных языков и грамматик, общая структура работы компилятора, теория автоматов. Пособие предназначено для самостоятельной работы студентов, в качестве дополнения к материалам лекций. Составлено в соответствии с требования государственного образовательного стандарта.

Рецензенты: Семочкин А.Н., доц. кафедры информатики и методики преподавания информатики БГПУ, канд. физ.-мат. наук;
Рыженко А.В., доц. кафедры математического анализа и моделирования АмГУ, канд. техн. наук.

ВВЕДЕНИЕ

Область системного программирования начала бурно развиваться в 60-х гг. прошлого столетия. С этого времени построены основные принципы программирования, созданы десятки новых языков программирования высокого уровня и компиляторов к ним. Но технологии, заложенные первоначально, современны и ныне.

В пособие уделено внимание теории формальных языков и грамматик, теории автоматов, которые служат основой, как для изучения работы компиляторов, так и для их реальной разработки.

Пособие предназначено для студентов специальности 230102 – Автоматизированные системы обработки информации и управления. При изучении одноименной дисциплины оно позволит повысить степень усвоения теоретического лекционного материала и эффективность самостоятельной работы студента. Для этого каждый раздел пособия завершается контрольными вопросами и заданиями для самостоятельного выполнения.

Материал пособия также может быть полезен в дисциплине «Лингвистические основы информатики» специальности 230201 – «Информационные системы и технологии».

Языки и грамматики

Языки программирования занимают некоторое промежуточное положение между формальными и естественными языками. С формальными языками их объединяют строгие синтаксические правила, на основе которых строятся предложения языка. От языков естественного общения в языки программирования перешли лексические единицы, представляющие основные ключевые слова. Кроме того, из алгебры языки программирования переняли основные обозначения математических операций.

Для задания языка программирования требуется:

1) определить множество допустимых символов языка;

2) определить множество правильных программ языка;

3) задать смысл для каждой правильной программы.

Первые два вопроса полностью или частично удается решить с помощью теории формальных языков.

Грамматика – описание способа построения предложений некоторого языка. Грамматика – математическая система, определяющая язык, т.е. – генератор цепочек языка.

Грамматику языка можно описать различными способами. Для синтаксических конструкций языков программирования можно использовать формальное описание грамматики, построенное на основе системы правил (или продукций).

Цепочкой символов называют произвольную упорядоченную конечную последовательность символов, записанных друг за другом. Количество символов в ней называют длиной цепочки.

Правило (или продукция) – упорядоченная пара цепочек символов (α, β). В правилах важен порядок цепочек, поэтому их чаще записывают в виде $\alpha \to \beta$. Такая запись читается как «α порождает β» или «β по определению есть α».

Грамматика языка программирования содержит правила двух типов: первые (определяющие синтаксические конструкции языка) довольно легко поддаются формальному описанию; вторые (определяющие семантические ограничения языка) обычно излагаются в неформальной форме.

Любое описание (или стандарт) языка программирования обычно состоит из двух частей:

1) формальное изложение правил построения синтаксических конструкций,

2) описание семантических правил на естественном языке.

Формально грамматика G определяется как четверка G(**VT, VN, P**, S), где

VT – множество терминальных символов или алфавит терминальных символов;

VN – множество нетерминальных символов или алфавит нетерминальных символов;

P – множество правил грамматики, вида $\alpha \rightarrow \beta$, где $\alpha \in (\mathbf{VN} \cup \mathbf{VT})^+$, $\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$;

S – целевой (начальный) символ грамматики $S \in \mathbf{VN}$.

Обозначение вида $\mathbf{V}^+$ означает множество без пустой цепочки, а обозначение $\mathbf{V}^*$ – множество, включающее пустую цепочку.

Алфавиты терминальных и нетерминальных символов грамматики не пересекаются: $\mathbf{VN} \cap \mathbf{VT}$ = пустое множество. Это значит, что каждый символ в грамматике может быть либо терминальным, либо нетерминальным, но не может быть терминальным и нетерминальным одновременно. Множество $\mathbf{V} = \mathbf{VN} \cup \mathbf{VT}$ называют полным алфавитом грамматики G.

Целевой символ грамматики – это всегда нетерминальный символ.

Множество терминальных символов **VT** содержит символы, которые входят в алфавит языка, порождаемого грамматикой. Как правило, символы из множества **VT** встречаются только в цепочках правых частей правил. Множество нетерминальных символов VN содержит символы, которые определяют слова, понятия, конструкции языка. Каждый символ этого множества может встречаться в цепочках как левой, так и правой частей правил грамматики, но он обязан хотя бы один раз быть в левой части хотя бы одного правила. Правила грамматики обычно строятся так, чтобы в левой части каждого правила был хотя бы один нетерминальный символ.

Во множестве правил грамматики может быть несколько правил, имеющих одинаковые левые части, вида: $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, ... $\alpha \rightarrow \beta_n$. Тогда эти правила объединяют вместе и записывают в виде: $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$. Одной строке в такой записи соответствует сразу n правил.

Такую форму записи правил грамматики называют формой Бэкуса-Наура. Форма Бэкуса-Наура предусматривает, как правило, также, что нетерминальные символы берутся в угловые скобки: < >.

Пример грамматики, которая определяет язык целых десятичных чисел со знаком:

G ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +}, {<число>, <чсл>, <цифра>}, P, <число>),

где P:

<число> → <чсл> | +<чсл> | –<чсл>

<чсл> → <цифра> | <чсл><цифра>

<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Составляющие элементы данной грамматики G:

множество терминальных символов **VT** содержит двенадцать элементов: десять десятичных цифр и два знака;

множество нетерминальных символов **VN** содержит три элемента: символы <число>, <чс> и <цифра>;

множество правил содержит 15 правил, которые записаны в три строки (то есть имеется только три различных правых части правил);

целевым символом грамматики является символ <число>.

Особенность рассмотренных формальных грамматик в том, что они позволяют определить бесконечное множество цепочек языка с помощью конечного набора правил (конечно, множество цепочек языка тоже может быть конечным, но даже для простых реальных языков это условие обычно не выполняется). Приведенная выше в примере грамматика для целых десятичных чисел со знаком определяет бесконечное множество целых чисел с помощью 15 правил.

В такой форме записи грамматики возможность пользоваться конечным набором правил достигается за счет рекурсивных правил. Рекурсия в правилах грамматики выражается в том, что один из нетерминальных символов определяется сам через себя.

Чтобы рекурсия не была бесконечной, для участвующего в ней нетерминального символа грамматики должны существовать также и другие правила, которые определяют его, минуя его самого, и позволяют избежать бесконечного рекурсивного определения (в противном случае этот символ в грамматике был бы просто не нужен). Такими правилами являются <чсл> → <цифра>.

Язык, заданный грамматикой G, обозначается как L(G).

Две грамматики G и G' называются эквивалентными, если они определяют один и тот же язык: L(G) = L(G'). Две грамматики G и G' называются почти эквивалентными, если заданные ими языки различаются не более чем на пустую цепочку символов: $L(G) \cup \{\lambda\} = L(G') \cup \{\lambda\}$, где λ – пустая цепочка.

Для компиляторов языки делятся на простые и сложные и существуют жесткие критерии для такого деления. Сложность построения компилятора зависит от сложности языка программирования, для которого он создается.

Согласно классификации, предложенной Ноамом Хомским, формальные грамматики классифицируются по структуре их правил. Если все без исключения правила грамматики удовлетворяют некоторой заданной структуре, то такую грамматику относят к определенному типу. По классификации Хомского выделяют четыре типа грамматик.

Тип 0: грамматики с фразовой структурой

На структуру их правил не накладывается никаких ограничений. Это самый общий тип грамматик. В него подпадают все без исключения формальные грамматики, но часть из них может быть также отнесена и к другим классификационным типам. Грамматики, которые относятся только к типу 0 и не могут быть отнесены к другим типам, являются самыми сложными по структуре. Практического применения грамматики, относящиеся только к типу 0, не имеют.

Тип 1: контекстно-зависимые (КЗ) и неукорачивающие грамматики

В этот тип входят два основных класса грамматик:

Контекстно-зависимые грамматики G(**VT,VN, P**, S), **V = VN**$\cup$ **VT** имеют правила вида: $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in$ V*, A$\in$ **VN**, $\beta \in$ **V**$^+$.

Неукорачивающие грамматики G**(VT,VN, P,** S**), V = VN**$\cup$ **VT** имеют правила вида: $\alpha \rightarrow \beta$, где $\alpha, \beta \in$ **V**$^+$, $|\beta| \geq |\alpha|$.

Структура правил КЗ-грамматик такова, что при построении предложений заданного ими языка один и тот же нетерминальный символ может быть заменен той или иной цепочкой символов в зависимости от контекста, в котором он встречается. Отсюда пошло и название «контекстно-зависимыми». Цепочки α_1 и α_2 в правилах грамматики обозначают контекст (α_1 – левый контекст, а α_2 – правый контекст), в общем случае любая из них (или даже обе) может быть пустой, т.е. значение одного и того же символа может быть различным в зависимости от контекста, в котором он встречается.

Неукорачивающие грамматики имеют такую структуру правил, что при построении предложений языка, заданного грамматикой, любая цепочка символов может быть заменена цепочкой символов не меньшей длины.

Доказано, что эти два класса грамматик эквивалентны. Это значит, что для любого языка, заданного контекстно-зависимой грамматикой, можно построить неукорачивающую грамматику, которая будет задавать эквивалентный язык и, наоборот: для любого языка, заданного неукорачивающей грамматикой, можно построить контекстно-зависимую грамматику, которая будет задавать эквивалентный язык.

При построении компиляторов такие грамматики не применяются, поскольку синтаксические конструкции языков программирования, рассматриваемые компиляторами, имеют более простую структуру.

Тип 2: контекстно-свободные (КС) грамматики

КС-грамматики G (**VT, VN, P**, S), **V = VN∪ VT** имеют правила вида: A→β , где A∈ **VN**, β ∈ **V**$^+$. Такие грамматики также иногда называют неукорачивающими контекстно-свободными (НКС) грамматиками (в правой части правила у них должен всегда стоять как минимум один символ).

Существует также почти эквивалентный им класс грамматик – укорачивающие контекстно-свободные (УКС) грамматики G (**VT, VN, P**, S), **V = VN∪ VT**, правила которых могут иметь вид: A→β , где A∈ **VN**, β ∈ **V***.

Разница между этими двумя классами грамматик заключается в том, что в УКС-грамматиках в правой части правил может присутствовать пустая цепочка (λ), а в НКС-грамматиках нет.

КС-грамматики широко используются при описании синтаксических конструкций языков программирования. Синтаксис большинства известных языков программирования основан именно на КС-грамматиках. Внутри типа КС-грамматик кроме классов НКС и УКС выделяют еще целое множество различных классов грамматик, и все они относятся к типу 2.

Тип 3: регулярные грамматики

Леволинейные грамматики G (**VT,VN, P**, S), **V = VN∪ VT** могут иметь правила ух видов: A→Bγ или A→γ , где A, B∈ **VN**, γ ∈ **VT*.**

Праволинейные грамматики G(**VT, VN, P**, S), **V = VN∪ VT** могут меть правила тоже двух видов: A→γ B или A→γ , где A, B∈ **VN**, β ∈ **VT***.

Эти два класса грамматик эквивалентны и относятся к типу регулярных грамматик.

Регулярные грамматики используются при описании простейших конструкций языков программирования: идентификаторов, констант, строк, комментариев т. д. Эти грамматики исключительно просты и удобны в использовании, поэтому в компиляторах на их основе строятся функции лексического анализа входного языка.

Типы грамматик соотносятся между собой особым образом. Из определения типов 2 и 3 видно, что любая регулярная грамматика является КС-грамматикой, но не наоборот. Также очевидно, что любая грамматика может быть отнесена к типу 0, поскольку он не накладывает никаких ограничений на правила. В то же время существуют укорачивающие КС-грамматики (тип 2), которые не являются ни контекстно-зависимыми, ни неукорачивающими (тип 1), поскольку могут содержать правила вида «$A \rightarrow \lambda$ », недопустимые в типе 1.

Одна и та же грамматика в общем случае может быть отнесена к нескольким классификационным типам. Для классификации грамматики всегда выбирают максимально возможный тип, к которому она может быть отнесена. Сложность грамматики обратно пропорциональна номеру типа, к которому относится грамматика. Грамматики, которые относятся только к типу 0, являются самыми сложными, а грамматики, которые можно отнести к типу 3, – самыми простыми.

Рассмотренная в примере грамматика, определяющая язык целых десятичных чисел со знаком относится к контекстно-свободным грамматикам (тип 2). Следовательно, ее можно отнести и к типу 0 и к типу 1. Данная грамматика не может быть отнесена к типу 3, поскольку правило <чсл> → <чсл><цифра> недопустимо для него.

Языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы. Сложность языка также убывает с возрастанием номера классификационного типа языка.

В основе большинства современных языков программирования лежат контекстно-свободные языки.

Контрольные вопросы

1. Как выглядит описание грамматики в форме Бэкуса-Наура?

2. Каковы составляющие формального описания грамматики?

3. Как классифицируются языки? Как их классификация соотносится с классификацией грамматик?

4. Почему язык программирования нельзя не является чисто формальным языком?

5. Какой тип грамматики самый сложный?

6. Грамматики G(**VT,VN, P**, S), **V = VN**$\cup$ **VT** , которые могут иметь правила двух видов: A $\rightarrow$ Bγ или A $\rightarrow$ γ , где A, B$\in$ **VN**, $\gamma \in$ **VT*** относятся к типу:

а) праволинейных грамматик;

б) леволинейных грамматик;

в) контексно-свободных грамматик;

г) контексно-зависимых грамматик.

7. Выберите верное утверждение:

а) неукорачивающие грамматики имеют такую структуру правил, что при построении предложений языка, заданного грамматикой, любая цепочка символов может быть заменена цепочкой символов меньшей длины;

б) целевой символ грамматики – это всегда терминальный символ;

в) языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы;

г) языки программирования являются формальными языками.

8. Для любого языка, заданного какой грамматикой, можно построить неукорачивающую грамматику, которая будет задавать эквивалентный язык.

а) контекстно-свободной грамматикой

б) грамматикой с фразовой структурой

в) контексно-зависимой грамматикой

г) регулярной грамматикой

Задание

1. Определить тип указанных грамматик

а) G1 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – }, {A, B}, P, A):

P:

A→ B | +B | –B

B→ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0B | 1B | 2B | 3B | 4B | 5B | 6B | 7B | 8B | 9B

б) G2 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – }, {S, B}, P, S):

P:

B→ + | – | λ

S→ B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | B9 | S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9

в) G3 ({0, 1}, {A, S}, P, S)

P:

S→ 0A1

0A→00A1

A→λ

г) G4 ({0, 1}, {A, S}, P, S)

P:

S→ 0A1 | 01

0A→00A1 | 001

A→λ

д) G5 ({0, 1}, { S}, P, S)

P:

S→ 0S1 | 01

е) G5 ({f, g, h}, { G, H, E, S}, P, S)

P:

S→ GH

G → fGgH | fg

Hg →gH

HE → Hh

gEh → ghh

fgE → fgh

2. Определить язык грамматики G ({+, – , *, /, (,), x, y}, {S}, P, S):

P:

S → S+S | S–S | S*S | S/S | (S) | x | y

3. Поездом называется произвольная последовательность локомотивов и вагонов. Построить грамматику в форме Бэкуса-Наура для понятия «поезд», если

а) поезд всегда начинается с локомотива;

б) все локомотивы должны быть сосредоточены в начале поезда;

в) поезд начинается с локомотива и заканчивается локомотивом;

г) в поезде должны чередоваться через два локомотивы и вагоны;

д) поезд не должен содержать два локомотива или два вагона подряд;

е) поезд не должен содержать подряд два локомотива.

Сентенциальная форма грамматики. Однозначность и эквивалентность грамматик

Цепочка $\beta = \delta_1\gamma\delta_2$ называется *непосредственно выводимой* из цепочки $\alpha = \delta_1\omega\delta_2$ в грамматике G**(VT,VN, P,** S**), V = VN∪ VT,** $\delta_1, \gamma, \delta 2 \in$ **V*****,** $\omega \in$ **V**$^+$, Если в грамматике G существует правило: $\omega \rightarrow \gamma \in$ **P**. Непосредственная выводимость цепочки β из цепочки α обозначается: $\alpha \Rightarrow \beta$. Из определения следует, что если взять несколько символов в цепочке α и заменить их на другие символы согласно некоторому правилу грамматики и получить цепочку β, то β непосредственно выводима из α.

Цепочка β называется *выводимой* из цепочки α (обозначается: α =>* β), в том случае, если выполнено одно из двух условий:

1. β непосредственно выводима из α (α => β);
2. существует такая γ, что γ выводима из α и β непосредственно выводима из γ (α =>* γ и γ => β).

Суть определения заключается в том, что α => β, то можно построить последовательность непосредственно выводимых цепочек от α к β, в которой каждая последующая цепочка непосредственно выводима из предыдущей цепочки. Такая последовательность называется выводом или цепочкой вывода.

Если цепочка вывода из α к β содержит одну и более промежуточных цепочек, она имеет специальное обозначение α =>+ β. Если количество шагов известно, можно его указать непосредственно у знака выводимости. Например, выражение $\alpha =>^3 \beta$ означает, что β выводится из α за три вывода.

Вывод называется окончательным, если на основе цепочки β, полученной в результате вывода, нельзя больше сделать ни одного шага вывода. Иначе говоря, вывод законченный, если цепочка, полученная в его результате пустая, либо содержит только терминальные символы грамматики.

Язык L, заданный грамматикой G**(VT, VN, P,** S**) –** это множество всех сентенциальных форм грамматики G. Язык L, заданный грамматикой G обозначается как L(G). Алфавитом языка L(G) будет множество терминальных символов грамматики **VT**, поскольку все конечные сентенциальные формы грамматики – цепочки над алфавитом **VT**. Тогда очевидно, что две эквивалентные грамматики должны иметь, по крайней мере, пересекающиеся множества терминальных символов. (Как правило, эти множества совпадают).

Вывод называется *левосторонним*, если в нем на каждом шаге вывода правило применяется к крайнему левому нетерминальному символу в цепочке. Аналогично, вывод называется *правосторонним*, если в нем на каждом шаге вывода правило применяется к крайнему правому нетерминальному символу в цепочке.

Примером левостороннего вывода является: S => Y => TY => TT => YTT => TTT, а правостороннего S => T => TX => T5 => T6. Вывод F => R является одновременно и лево- и правосторонним.

Встречаются выводы, которые нельзя отнести ни к левосторонним, ни к правосторонним, например TFT => TFFT => TFFF => FFFF => FFTFF.

Для грамматик типов 2 и 3 (КС-грамматик и регулярных грамматик) всегда можно построить левосторонний или правосторонний выводы. Для грамматик других типов это не всегда возможно, так как по структуре их правил не всегда можно выполнить замену крайнего левого или крайнего правого нетерминального символа в цепочке.

Деревом вывода грамматики G**(VT,VN, P,** S**),** называется граф, который соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

каждая вершина графа обозначается некоторым символом грамматики A$\in$ **VN**$\cup$ **VT;**

корнем дерева является вершина, обозначенная целевым символом грамматики S;

листьями дерева являются вершины, обозначенные терминальными символами грамматики или символом пустой цепочки λ ,

если некоторый узел дерева обозначен нетерминальным символом A$\in$ **VN**, а связанные с ним узлы – символами b, b_2... b_n; n > 0, 0 < i < n, $b_i\in$ (**VN**$\cup$ **VT**$\cup$ {λ }), то в грамматике G**(VT,VN, P,** S**),** существует правило A $\rightarrow$ $b_1b_2\ldots b_n\in$ P.

Из определения следует, что по структуре правил дерево вывода в указанном виде всегда можно построить только для грамматик типов 2 и 3 (контекстно-свободных и регулярных). Для грамматик других типов дерево вывода в таком виде можно построить не всегда (либо же оно будет иметь несколько иной вид).
Для того чтобы построить дерево вывода, достаточно иметь только цепочку вывода. Дерево вывода можно построить двумя способами: сверху вниз и снизу вверх. Для строго формализованного построения дерева вывода всегда удобнее пользоваться строго определенным выводом: либо левосторонним, либо правосторонним.

При построении дерева вывода сверху вниз построение начинается с целевого символа грамматики, который помещается в корень дерева. Затем в грамматике выбирается необходимое правило, и на первом шаге вывода корневой символ раскрывается на несколько символов первого уровня. На втором шаге среди всех концевых вершин дерева выбирается крайняя (крайняя левая – для левостороннего вывода, крайняя правая – для правостороннего) вершина, обозначенная нетерминальным символом, для этой вершины выбирается нужное правило грамматики, и она раскрывается на несколько вершин следующего уровня. Построение дерева заканчивается, когда все концевые вершины обозначены терминальными символами, в противном случае надо вернуться ко второму шагу и продолжить построение.

Построение дерева вывода снизу вверх начинается с листьев дерева. В качестве листьев выбираются терминальные символы конечной цепочки вывода, которые на первом шаге построения образуют последний уровень (слой) дерева. На втором шаге в грамматике выбирается правило, правая часть которого соответствует крайним символам в слое дерева (крайним правым символам при правостороннем выводе и крайним левым при левостороннем). Выбранные вершины слоя соединяются с новой вершиной, которая выбирается из левой части правила. Новая вершина попадает в слой дерева вместо выбранных вершин. Построение дерева закончено, если достигнута корневая вершина (обозначенная целевым символом), а иначе надо вернуться ко второму шагу и повторить его относительно полученного слоя дерева.

Все известные языки программирования имеют нотацию записи «слева направо», компилятор также всегда читает входную программу слева направо (и сверху вниз, если программа разбита на несколько строк). Поэтому для построения дерева вывода методом «сверху вниз», как правило, используется левосторонний вывод, а для построения «снизу вверх» – правосторонний вывод.

Грамматика называется *однозначной,* если для каждой цепочки символов языка, заданного этой грамматикой, можно построить единственный левосторонний (и единственный правосторонний) вывод. Грамматика также называется однозначной, если для каждой цепочки символов языка, заданного этой грамматикой, существует единственное дерево вывода. В противном случае грамматика называется *неоднозначной.*

Рассмотрим некоторую грамматику G ({+, – , *, /, (,), x, y}, {S}, P, S):

P:

S → S+S | S–S | S*S | S/S | (S) | x | y

Грамматика определяет язык арифметических выражений с четырьмя основными операциями: сложение, вычитание, умножение, деление и скобками.

Для цепочки, принадлежащей данному языку, x*y+x можно построить два варианта левостороннего вывода:

S => S+S => S*S+S => x*S+S => x*y+S => x*y+x

S => S*S => x*S => x*S+S => x*y+S => x*y+S

Каждому из этих вариантов будет соответствовать свое дерево вывода (рис. 1).

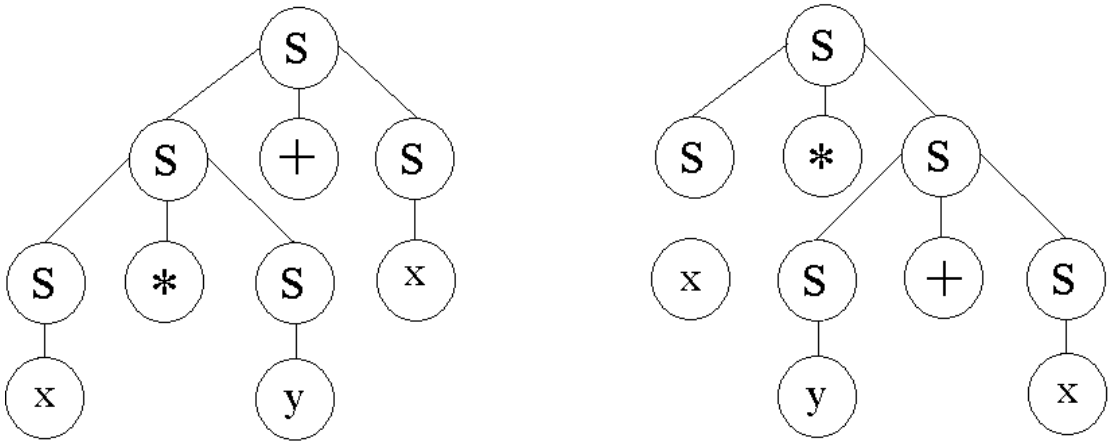


Рис. 1. Варианты дерева выражения x*y+x

Дерево вывода (или цепочка вывода) является формой представления структуры предложения языка. Поэтому для языков программирования, которые несут смысловую нагрузку, имеет принципиальное значение то, какая цепочка вывода будет построена для того или иного предложения языка. В рассмотренной грамматике все операции равноправны и для них не определен порядок выполнения. Поэтому с точки зрения арифметических операций приведенная грамматика имеет неверную семантику, хотя синтаксическая структура построенных с ее помощью выражений будет правильной. Такая ситуация вызывает неоднозначность в грамматике.

Если грамматика является неоднозначной, необходимо попытаться преобразовать ее в однозначный вид. Например, для рассмотренной грамматики арифметических выражений существует эквивалентная ей однозначная грамматика вида:

G’ ({+, – , *, /, (,), x, y}, {S}, P’, S):

P’:

S → S+T | S–T | T

T→ T*E | T/E | E

E→ (S) | x | y

Для арифметического выражения x*y+x в этой грамматике можно построить единственный левосторонний вывод:

S => S+T => T+T => T*E+T => E*E+T => x*E+T => x*y+T => x*y+E => x*y+x

На рис. 2 представлено единственно возможное дерево соответствующее этому выводу.

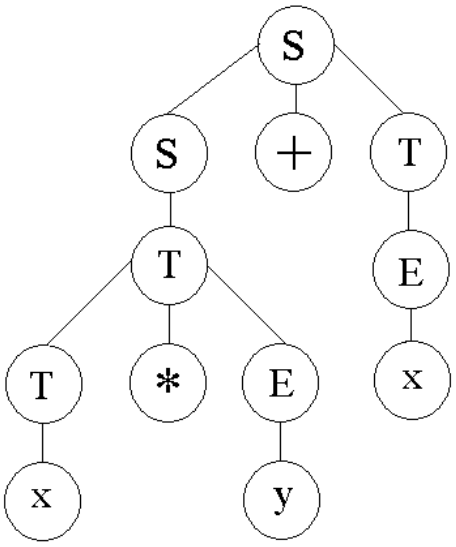


Рис. 2. Дерево вывода выражения x*y+x для однозначной грамматики

К сожалению, доказано, что не существует алгоритма, позволяющего проверить однозначность и эквивалентность грамматик. Однако, неразрешимость проблем эквивалентности и однозначности грамматик в общем случае не означает, что они не разрешимы вообще. Для многих частных случаев эти проблемы решены. Например, для КС-грамматик существуют правила определенного вида, по наличию которых во всем множестве правил грамматики G**(VT, VN, P,** S**)** можно утверждать, что она является неоднозначной. Эти правила имеют следующий вид:

1) $A \rightarrow AA \mid \alpha$

2) $A \rightarrow A\alpha A \mid \beta$

3) $A \rightarrow \alpha A \mid A\beta \mid \gamma$

4) $A \rightarrow \alpha A \mid \alpha A \beta A \mid \gamma$

здесь $A \in$ **VN**; $\alpha, \beta, \gamma \in$ **(VN** $\cup$ **VT)***.

Если в КС-грамматике встречается хотя бы одно правило любого из приведенных вариантов, то доказано, что такая грамматика точно будет неоднозначной. Однако если подобных правил во всем множестве правил грамматики нет, это совсем не означает, что грамматика является однозначной. То есть отсутствие правил указанного вида – необходимое, но не достаточное условие однозначности грамматики.

Существуют условия, при выполнении которых грамматика заведомо является однозначной. Они справедливы для всех регулярных и многих классов контекстно-свободных грамматик. Эти условия, напротив, являются достаточными, но не необходимыми для однозначности грамматик.

Контрольные вопросы

1. Что такое сентенциальная форма грамматики?
2. В чем заключается отличие левосторонних и правосторонних выводов?
3. В чем заключается однозначность грамматики?
4. Дайте определение полностью выводимой цепочки.
5. По каким правилам строится дерево вывода?
6. Выберите неверное утверждение:

а) однозначность – это свойство грамматики;

б) построение дерева вывода заканчивается, когда все концевые вершины обозначены терминальными символами;

в) проблема эквивалентности грамматик разрешима алгоритмически;

г) для того чтобы построить дерево вывода достаточно иметь только цепочку вывода.

Задание

1. Дана грамматика G**({“ ”,** i, f, t, h, e, n, l, s, b, a**}**, {E}, P**,** E**)** с правилами:

P:

E → if b then E else E; | if b then E; | a

Не строя цепочек вывода, показать, что грамматика является неоднозначной. Проверить это, построив некоторую цепочку вывода.

2. Построить эквивалентную ей однозначную грамматику. Построить для нее дерево вывода.

3. Указать к какому типу относится каждая из грамматик языка десятичных чисел с фиксированной точкой следующие грамматики:

а) G1 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – , “.”}, {<число>, <цел>, <дроб>, <цифра>, <осн>, <знак>}, P1, <число>)

P1:

<число> → <знак> <осн>

<знак> → λ | + | -

<осн> → <цел>. <дроб> | <цел>.

<цел> → <цифра> | <цифра> <цифра>

<дроб> → λ | <цел>

<цифра> <цифра> → <цифра> <цифра> <цифра>

<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7| 8 | 9

б) G2 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – , “.”}, {<число>, <часть>, <цифра>, <осн>}, P2, <число>)

P2:

<число> → +<осн>| -<осн> | <осн>

<осн> → <часть>.<часть> |<часть>. |<часть>

<часть> → <цифра> | <цифра> <цифра>

<цифра> <цифра> → <цифра> <цифра> <цифра>

<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7| 8 | 9

в) G3 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – , “.”}, {<число>, <часть>, <осн>}, P3, <число>)

P3:

<число> → +<осн>| -<осн> | <осн>

<осн> → <часть>. | <часть> | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7| <осн>8 | <осн>9

<часть> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7| 8 | 9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7| <часть>8 | <часть>9

4. Определить является ли однозначной каждая из грамматик, описанная в задание 3.

5. Построить цепочки вывода для цепочек -57, 196, 11.9 но основе грамматик из задания 3.

6. Перестроить грамматики, определенные в задании 3 таким образом, чтобы они допускали цепочки вида: .124, .34.

Основные принципы построения компиляторов

Транслятор – программа, которая считывает текст программы на исходном языке и транслирует (переводит) ее в эквивалентный текст на другом выходном языке. Важным моментом трансляции является сообщение пользователю о наличие ошибок в исходной программе (рис.3).

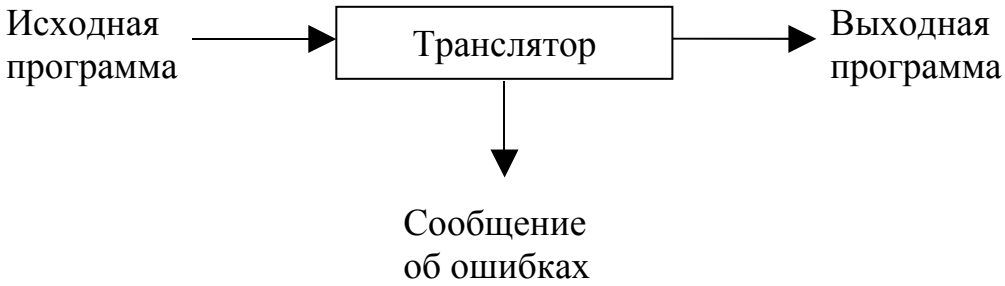


Рис.3. Работа транслятора

Близким по смыслу к понятию «транслятор» является понятие «компилятор». Компилятор – это транслятор, осуществляющий перевод исходной программы в эквивалентную ей результирующую программу на языке машинных команд или на языке ассемблера. Компиляторы самый распространенный тип трансляторов.

Компиляция состоит из двух частей анализа и синтеза. Анализ – это разбиение исходной программы на составные части и создание ее промежуточного представления. Синтез – конструирование требуемой выходной программе. В каждой фазе компиляции могут встретиться ошибки. После их обнаружения необходимы определенные действия для выявления других ошибок в исходной программе.

На рис.4 представлена общая схема работы компилятора.

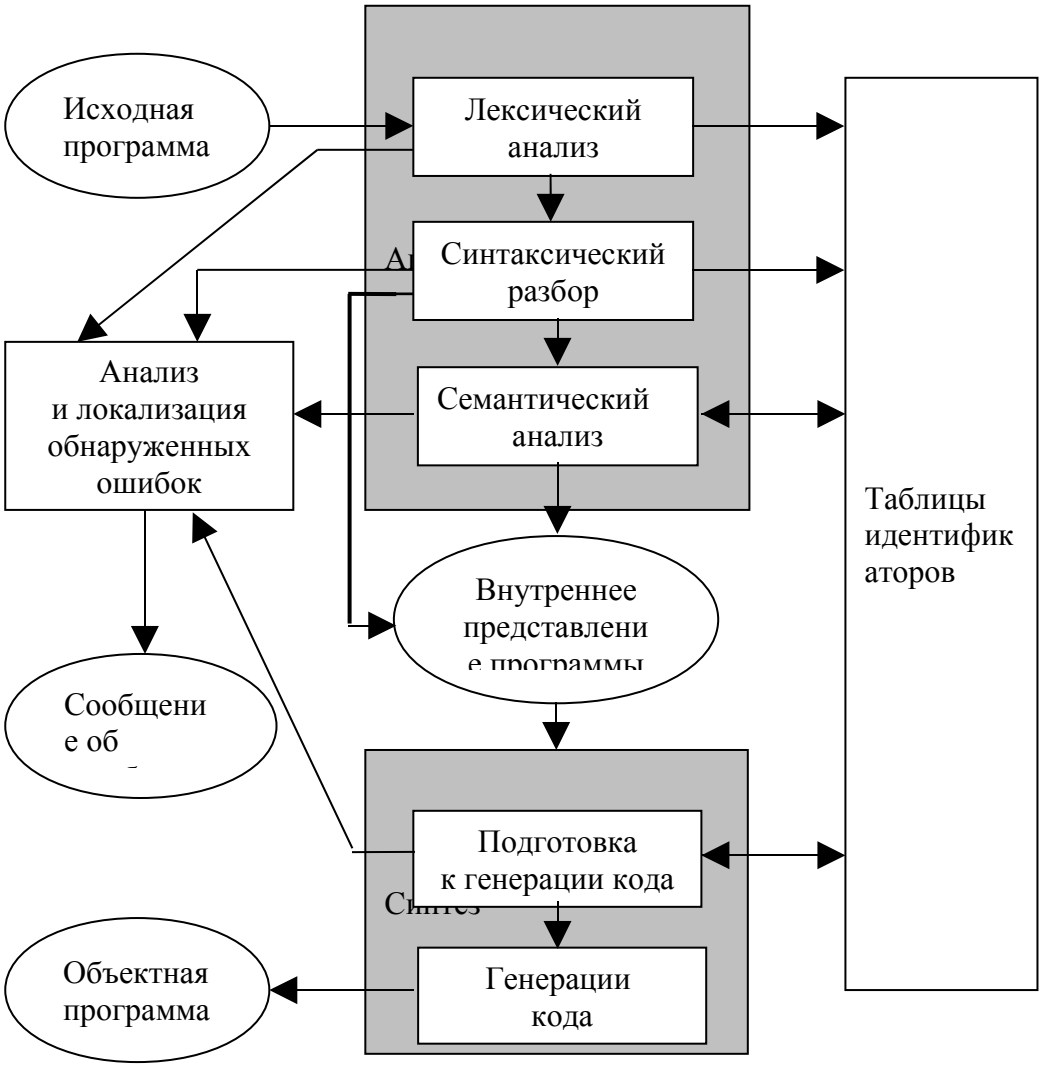


Рис. 4. Общая схема работы компилятора

Анализ состоит из трех фаз. Основная задача лексического анализа – разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов или лексем, то есть выделить эти слова из непрерывной последовательности символов. Все символы входной последовательности с этой точки зрения разделяются на принадлежащие каким-либо лексемам и на разделяющие лексемы (разделители).

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет, могут меняться в зависимости от реализации компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев, незначащих пробелов, символов табуляции и перевода строки и выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых слов, знаков операций.

Результатом его работы является перечень всех найденных в тексте исходной программы лексем с учетом характеристик каждой из них. Этот перечень можно представить в виде таблицы, называемой таблицей лексем. Каждой лексеме в ней соответствует некий уникальный условный код, зависящий от типа лексемы, и дополнительная служебная информация.

Информация о некоторых типах лексем должна помещаться в таблицу идентификаторов. Любая лексема может встречаться в таблице лексем любое количество раз.

Синтаксический разбор – основная часть компилятора на этапе анализа, выполняющая выделение синтаксических конструкций в тексте исходной программы, обработанном лексическим анализатором, т.е. группирование выделенных лексем в грамматические фразы.

Семантический анализ – это часть компилятора, проверяющая правильность текста исходной программы с точки зрения семантики входного языка. Кроме непосредственной проверки, выполняется преобразования текста, требуемые семантикой входного языка (например, добавление функций неявного преобразования типов). В различных реализациях компиляторов семантический анализ может частично входить в фазу синтаксического разбора, частично — в фазу подготовки к генерации кода.

Подготовка к генерации кода – фаза, на которой компилятором выполняются предварительные действия, непосредственно связанные с синтезом текста результирующей программы, но еще не ведущие к порождению текста на выходном языке. Обычно в эту фазу входят действия, связанные с идентификацией элементов языка, распределением памяти и т. п.

Генерация кода – фаза, непосредственно связанная с порождением команд, составляющих предложения выходного языка и в целом текст результирующей программы. Это основная фаза на этапе синтеза результирующей программы. Кроме непосредственного порождения текста результирующей программы генерация обычно включает в себя оптимизацию – процесс, связанный с обработкой уже порожденного текста. Иногда оптимизацию выделяют в отдельную фазу компиляции, так как она оказывает существенное влияние на качество и эффективность результирующей программы.

Таблицы идентификаторов (таблицы символов) – это специальным образом организованные наборы данных, служащие для хранения информации об элементах исходной программы, которые затем используются для порождения текста результирующей программы. В конкретной реализации компилятора может быть как одна, так и несколько таблиц идентификаторов. Элементами исходной программы, информацию о которых необходимо хранить в процессе компиляции, являются переменные, константы, функции и т. п. (конкретный состав набора элементов зависит от используемого входного языка программирования). Понятие «таблицы» не предполагает, что это хранилище должно быть организовано в виде таблиц или других массивов информации.

Представленное на рисунке деление процесса компиляции на фазы служит методическим целям и на практике может столь строго не соблюдаться.

С точки зрения формальных языков компилятор является распознавателем языка исходной программы и генератором для языка выходной программы.

Реальные компиляторы выполняют трансляцию текста исходной программы, как правило, за несколько проходов. *Проходом* называют процесс последовательного чтения данных из внешней памяти, их обработку и помещение результата работы обратно во внешнюю память. Как правило, один проход включает в себя одну или несколько фаз компиляции.

Интерпретатор – программа, воспринимающая исходную программу на исходном языке и выполняющая ее. Интерпретатор не порождает результирующую программу.

Контрольные вопросы

1. Что такое компилятор?
2. Перечислить основные этапы анализа компилятора?
3. Назовите отличия между транслятором, компилятором, интерпретатором.
4. Как называется программа, которая воспринимает исходную программу на входном (исходном) языке и выполняет ее?
5. Как называется часть компилятора, проверяющая правильность текста исходной программы с точки зрения смысла входного языка?
6. Как могут быть связаны между собой лексический и синтаксический анализы?

Задание

1. Построить лексический сканер, который выделял бы из текста входной программы, написанной на языке C++ все, содержащиеся в ней, константы целых типов. Записать их в отдельный файл. Не забыть игнорировать комментарии. Алгоритм выделения лексем представить в виде блок-схемы.
2. Построить лексический сканер, который выделял бы из текста входной программы, написанной на языке Pascal все, содержащиеся в ней, идентификаторы. Записать их в отдельный файл. Не забыть игнорировать комментарии. Алгоритм выделения лексем представить в виде блок-схемы.
3. Построить лексический сканер, который выделял бы из текста входной программы, написанной на языке C++ все, знаки операций. Записать их в отдельный файл. Не забыть игнорировать комментарии. Алгоритм выделения лексем представить в виде блок-схемы.

4. Построить лексический сканер, который выделял бы из текста входной программы, написанной на языке Pascal, все, содержащиеся в ней, знаки операций. Записать их в отдельный файл. Не забыть игнорировать комментарии. Алгоритм выделения лексем представить в виде блок-схемы.

5. Построить лексический сканер, который выделял бы из текста входной программы, написанной на языке C++ все, содержащиеся в ней, ключевые слова. Записать их в отдельный файл. Не забыть игнорировать комментарии. Алгоритм выделения лексем представить в виде блок-схемы.

6. Построить лексический сканер, который выделял бы из текста входной программы, написанной на языке C++ все, содержащиеся в ней, ключевые слова. Записать их в отдельный файл. Не забыть игнорировать комментарии. Алгоритм выделения лексем представить в виде блок-схемы.

Методы организация таблиц идентификаторов

Выделение идентификаторов и других элементов исходной программы происходит на фазе лексического анализа. Для хранения найденных идентификаторов и их характеристик используются специальные хранилища данных, называемые *таблицами символов,* или *таблицами идентификаторов.*

Любая таблица идентификаторов состоит из набора полей, количество которых равно числу различных идентификаторов, найденных в исходной программе. Каждое поле содержит в себе полную информацию о данном элементе таблицы. Компилятор может работать как с одной, так и с несколькими таблицам идентификаторов.

Состав информации, хранимой в таблице идентификаторов для каждого элемента исходной программы, зависит от семантики входного языка и типа элемента. Так для переменных могут храниться: имя переменной, тип данных и область памяти, связанная с переменной; для функций – имя функции, количество и типы формальных аргументов функции, тип возвращаемого результата; адрес кода функции.

Не вся информация, хранимая в таблице идентификаторов, заполняется компилятором одновременно. Имена переменных могут быть выделены на фазе лексического анализа, типы данных для них – на фазе синтаксического разбора, а область памяти – на фазе подготовки к генерации кода. Таким образом, на разных фазах компиляции компилятор многократно обращается к таблице для поиска информации и записи новых данных.

Таблицы идентификаторов организуются таким образом, чтобы компилятор имел возможность максимально быстрого поиска требуемого ему элемента.

Простейший способ ее организации состоит в добавлении новых элементов в порядке их поступления. В этом случае таблица идентификаторов представляет собой неупорядоченный массив информации. Поиск нужного элемента заключается в последовательном сравнении искомого элемента с каждым элементом таблицы. Тогда для поиска в таблице, содержащей n элементов, в среднем будет выполнено n/2 сравнений. Такой способ организации таблиц идентификаторов является неэффективным.

Поиск более эффективен в таблице, элементы которой упорядочены (отсортированы) согласно некоторому порядку. Методом поиска в упорядоченном списке является *бинарный* (или *логарифмический*) *поиск.*

Его алгоритм состоит в следующем: искомый символ сравнивается с элементом в середине таблицы (с порядковым номером (N + 1)/2). Если этот элемент не является искомым, то просматривается только блок элементов, пронумерованных от 1 до (N + 1)/2 – 1, или блок элементов от (N + 1)/2 + 1 до N в зависимости от того, меньше или больше искомый элемент по сравнению с ранее найденным. Так продолжается до тех пор, пока либо искомый элемент не будет найден, либо алгоритм дойдет до очередного блока, содержащего один или два элемента, с которыми можно выполнить прямое сравнение искомого элемента.

Так как на каждом шаге число элементов, которые могут содержать искомый элемент, сокращается в 2 раза, максимальное число сравнений равно $1+\log_2(N)$.

Недостатком данного метода является требование упорядочивания элементов таблицы идентификаторов. Время упорядочивания напрямую зависит от числа элементов в массиве. Таблица идентификаторов зачастую просматривается компилятором еще до того, как она заполнена полностью, поэтому для построения такой таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.

Для сокращения времени поиска искомого элемента в таблице идентификаторов без значительного увеличения времени ее заполнения, надо отказаться от организации таблицы в виде непрерывного массива данных.

Например, существует метод построения таблиц в форме бинарного дерева. Каждый узел такого дерева представляет собой элемент таблицы, причем корневой узел является первым элементом, встреченным при заполнении таблицы. Дерево называется бинарным, так как каждая вершина в нем может иметь не более двух ветвей. Для определенности их называют «правая» и «левая».

Первый идентификатор помещается в вершину дерева. Все дальнейшие идентификаторы попадают в дерево по следующему алгоритму:

Шаг 1. Выбрать очередной идентификатор из входного потока данных. Если его нет – построение дерева закончено.

Шаг 2. Сделать текущим узлом дерева корневую вершину.

Шаг 3. Сравнить очередной идентификатор с идентификатором, содержащимся в текущем узле дерева.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, если равен – сообщить об ошибке и прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно!), иначе перейти к шагу 7.

Шаг 5. Если у текущего узла существует левая вершина, сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 6.

Шаг 6. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

Шаг 7. Если у текущего узла существует правая вершина, сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 8.

Шаг 8. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину правой вершиной текущего узла и вернуться к шагу 1.

Поиск нужного элемента в дереве выполняется по алгоритму, схожему с алгоритмом заполнения дерева:

Шаг 1. Сделать текущим узлом дерева корневую вершину.

Шаг 2. Сравнить искомый идентификатор с идентификатором, содержащимся в текущем узле дерева.

Шаг 3. Если идентификаторы совпадают, искомый идентификатор найден, алгоритм завершен, иначе надо перейти к шагу 4.

Шаг 4. Если очередной идентификатор меньше, перейти к шагу 5, иначе перейти к шагу 6.

Шаг 5. Если у текущего узла существует левая вершина, сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершен.

Шаг 6. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершен.

Для данного метода число требуемых сравнений и форма дерева зависят от порядка, в котором поступают идентификаторы. Недостатком метода является необходимость работы с динамическим выделением памяти при построении дерева.

В целом метод бинарного дерева является довольно удачным механизмом для организации таблиц идентификаторов. Он нашел свое применение в ряде компиляторов. Иногда компиляторы строят несколько различных деревьев для идентификаторов разных типов и разной длины.

Лучших результатов можно достичь, если применить методы, связанные с использованием хэш-функций и хэш-адресации.

Хэш-функцией F называется некоторое отображение множества входных элементов R на множество целых неотрицательных чисел Z: $F(r) = n$, $r \in R$, $n \in Z$. Множество допустимых входных элементов R называется областью определения хэш-функции. Множеством значений хэш-функции F называется подмножество M из множества целых неотрицательных чисел Z: $M \subseteq Z$, содержащее все возможные значения, возвращаемые функцией F: $\forall r \in R: F(r) \in M$ и $\forall m \in M: \exists r \in R: F(r) = m$. Процесс отображения области определения хэш-функции на множество значений называется «хэшированием».

При работе с таблицей идентификаторов хэш-функция выполняет отображение имен идентификаторов на множество целых неотрицательных чисел. Областью определения такой хэш-функции будет множество всех возможных имен идентификаторов.

Хэш-адресация заключается в использовании значения, возвращаемого хэш-функцией, в качестве адреса ячейки из некоторого массива данных. Тогда размер массива данных должен соответствовать области значений используемой хэш-функции. Следовательно, в реальном компиляторе область значений хэш-функции не должна превышать размер доступного адресного пространства компьютера.

Метод организации таблиц идентификаторов, основанный на использовании хэш-адресации, заключается в размещении каждого элемента таблицы в ячейке, адрес которой возвращает хэш-функция, вычисленная для этого элемента. Для помещения каждого элемента в таблицу идентификаторов требуется вычисление его хэш-функции и размещения его по вычисленному адресу. Первоначально таблица идентификаторов должна содержать пустые ячейки.

Для поиска требуемого элемента в таблице также вычисляется хэш-функция и проверяется содержимое соответствующей ячейки. Если она не пуста – элемент найден, иначе – не найден.

Время размещения элемента в таблице и время его поиска определяются только временем, затрачиваемым на вычисление хэш-функции, которое в общем случае несопоставимо меньше времени, необходимого для выполнения многократных сравнений элементов таблицы. Но метод имеет два очевидных недостатка. Первый из них неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать области значений хэш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше. Второй – необходимость соответствующего разумного выбора хэш-функции.

Если двум или более идентификаторам соответствует одно и то же значение функции, такая ситуация называется *коллизией.* Хэш-функция, допускающая хотя бы единичную коллизию, не может быть напрямую использована для хэш-адресации в таблице идентификаторов.

Для полного исключения коллизий хэш-функция должна быть взаимно однозначной, т.е. каждому элементу из области определения хэш-функции должно соответствовать только одно значение из ее множества значений, и каждому значению из множества значений этой функции должен соответствовать только один элемент из области ее определения.

В реальности область значений любой хэш-функции ограничена размером доступного адресного пространства в данной архитектуре компьютера. Множество адресов любого компьютера с традиционной архитектурой может быть велико, но всегда конечно. Организовать взаимно однозначное отображение бесконечного множества имен идентификаторов на конечное множество невозможно.

Существует несколько способов для разрешения проблемы коллизии. Одним из них является метод *рехэширования* (расстановки). В нем, если для элемента A адрес h(A), вычисленный с помощью хэш-функции h, указывает на уже занятую ячейку, то необходимо вычислить новое значение $n_1 = h_1(A)$ и проверить занятость ячейки по адресу n_1. Если и она занята, то вычисляется значение $h_2(A)$ и так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение $h_i(A)$ совпадет с h(A). В последнем случае считается, что таблица идентификаторов заполнена, и места в ней больше нет.

Тогда таблицу идентификаторов можно организовать по следующему алгоритму размещения элемента:

Шаг 1. Вычислить значение хэш-функции n = h(A) для нового элемента A.

Шаг 2. Если ячейка по адресу n пустая, поместить в нее элемент A и завершить алгоритм, иначе i = 1 и перейти к шагу 3.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая, то поместить в нее элемент A и завершить алгоритм, иначе перейти к шагу 4.

Шаг 4. Если $n = n_i$ то сообщить об ошибке и завершить алгоритм, иначе i = i+1 и вернуться к шагу 3.

Поиск элемента A в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1. Вычислить значение хэш-функции n= h(A) для искомого элемента A.

Шаг 2. Если ячейка по адресу n пуста, то элемент не найден, алгоритм завершен. Иначе сравнить имя элемента в ячейке n с именем искомого элемента A. Если они совпадают – элемент найден и алгоритм завершен, иначе i = 1, перейти к шагу 3.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая или $n = n_i$ то элемент не найден и алгоритм завершен, иначе сравнить имя элемента в ячейке n_i с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершен, иначе i = i + 1 и повторить шаг 3.

Алгоритмы размещения и поиска элемента схожи по выполняемым операциям и имеют одинаковые оценки времени, необходимого для их выполнения.

При такой организации таблиц идентификаторов при возникновении коллизии алгоритм пытается поместить элемент в пустую ячейку, что может привести к возникновению новой, дополнительной коллизии. Поэтому количество операций, необходимых для поиска или размещения в таблице элемента, зависит от степени заполнения таблицы.

Важно определить хэш-функцию h_i для каждого i. Чаще всего функции h_i определяют как некоторые модификации первоначальной хэш-функции h. Например, самым простым методом вычисления функции $h_i(A)$ является ее организация в виде $h_i(A) = (h(A) + p_i) \bmod N_m$, где p_i – некоторое вычисляемое целое число, а N_m – максимальное значение из области значений хэш-функции h. Простейшим случаем будет задать $p_i = i$. Тогда при совпадении значений хэш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной значением хэш-функции h(A). В этом случае при совпадении хэш-адресов элементы в таблице начинают группироваться вокруг них, что увеличивает число необходимых сравнений при их поиске и размещении.

Но даже такой примитивный метод рехэширования является достаточно эффективным средством организации таблиц идентификаторов при частном заполнении таблицы. Имея, например, заполненную на 90% таблицу для 1024 идентификаторов, в среднем необходимо выполнить 5.5 сравнений для поиска одного идентификатора, в то время как даже логарифмический поиск дает в среднем от 9 до 10 сравнений.

Лучшие результаты дает использование в качестве p_i последовательности псевдослучайных целых чисел p1, p_2, ..., p_k или при вычислении по формуле $h_i(A) = (h(A)*i) \bmod N_m$, если N_m – простое число. В целом, рехэширование позволяет добиться неплохих результатов, но требование частичного заполнения таблицы ведет к неэффективному использованию объема доступной памяти.

Частичное заполнение таблицы идентификаторов при применении хэш-функций ведет к неэффективному использованию всего объема памяти, доступного компилятору. Этого недостатка можно избежать, дополнив таблицу идентификаторов специальной промежуточной хэш-таблицей. В ее ячейках может храниться либо пустое значение, либо значение указателя на некоторую область памяти из основной таблицы идентификаторов. Тогда после вычисления значения хэш-функции определяется адрес, по которому происходит обращение сначала к промежуточной хэш-таблице, а через нее – к самой таблице идентификаторов. Тогда иметь в самой таблице идентификаторов ячейку для каждого возможного значения хэш-функции не обязательно и таблицу можно сделать динамической. Количество ячеек в ней будет равно числу идентификаторов. Пустые ячейки будут только в хэш-таблице. Способ реализации такой схемы называется «метод цепочек». Он работает по следующему алгоритму:

Шаг 1. Во все ячейки хэш-таблицы поместить пустое значение, таблица идентификаторов пуста, переменная FreePtr (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; i = 1.

Шаг 2. Вычислить значение хэш-функции n_i для нового элемента A_i. Если ячейка хэш-таблицы по адресу n_i пустая, поместить в нее значение переменной FreePtr и перейти к шагу 5; иначе перейти к шагу 3.

Шаг 3. Положить j=1, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m_j и перейти к шагу 4.

Шаг 4. Для ячейки таблицы идентификаторов по адресу m_j проверить значение поля ссылки. Если оно пустое, записать в него адрес из переменной FreePtr и перейти к шагу 5; иначе j = j + 1, выбрать из поля ссылки адрес m_j и повторить шаг 4.

Шаг 5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A_i (поле ссылки должно быть пустым), в переменную FreePtr поместить адрес, следующий за добавленной ячейкой. Если больше нет идентификаторов для размещения в таблице, алгоритм завершен, иначе i = i + 1 и перейти к шагу 2.

Поиск элемента в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1. Вычислить значение хэш-функции n для искомого элемента A. Если ячейка хэш-таблицы по адресу n пустая, то элемент не найден и алгоритм завершен, иначе j = 1, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m_j.

Шаг 2. Сравнить имя элемента в ячейке таблицы идентификаторов по адресу m_j с именем искомого элемента A. Если они совпадают, искомый элемент найден и алгоритм завершен, иначе перейти к шагу 3.

Шаг 3. Проверить значение поля ссылки в ячейке таблицы идентификаторов по адресу m_j. Если оно пустое, то искомый элемент не найден и алгоритм завершен; иначе j = j + 1, выбрать из поля ссылки адрес m_j и перейти к шагу 2.

В случае возникновения коллизии алгоритм размещает элементы в ячейках таблицы, связывая их друг с другом последовательно через поле ссылки. При этом элементы не могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хэш-функции. Таким образом, дополнительные коллизии не возникают. В итоге в таблице возникают своеобразные цепочки связанных элементов, откуда происходит и название данного метода.

В реальных компиляторах практически всегда, так или иначе, используется хэш-адресация. Часто применяются комбинированные методы. В этом случае, как и для метода цепочек, в таблице идентификаторов организуется специальное дополнительное поле ссылки. Но в отличие от метода цепочек оно имеет несколько иное значение. При отсутствии коллизий для выборки информации из таблицы используется хэш-функция, поле ссылки остается пустым. Если же возникает коллизия, го с помощью поля ссылки организуется поиск идентификаторов, для которых значения хэш-функции совпадают, по одному из рассмотренных выше методов: неупорядоченный список, упорядоченный список или же бинарное дерево. При хорошо построенной хэш-функции коллизии будут возникать редко.

Хэш-адресация – метод, который применяется не только для организации таблиц идентификаторов в компиляторах, но и нашел свое применение в операционных системах, и в системах управления базами данных.

Контрольные вопросы

1. Какая информация хранится в таблице идентификаторов?
2. Какие способы организации таблиц идентификаторов существуют?
3. Когда и по какой причине возникает коллизия при организации таблиц идентификаторов с использованием хэш-функции?
4. Каковы требования к списку идентификаторов при использовании метода логарифмического поиска в таблице идентификаторов?
5. В чем заключается преимущество метода цепочек по сравнению с методом рехэширования?
6. Выберите неверное утверждение:

а) область значений любой хэш-функции ограничена размером доступного адресного пространства в данной архитектуре компьютера;

б) вся информация, хранимая в таблице идентификаторов, заполняется компилятором одновременно;

в) для полного исключения коллизий хэш-функция должна быть взаимно однозначной;

г) состав информации, хранимой в таблице идентификаторов для каждого элемента исходной программы, зависит от семантики входного языка и типа элемента.

7. Использование значения, возвращаемого хэш-функцией, в качестве адреса ячейки из некоторого массива данных называется

а) хэш-адресацией б) хэш-функцией

в) хэшированием г) рехэшированием

Задание

1. Написать программу, реализующую создание таблицы идентификаторов по методу логарифмического поиска. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами.

2. Написать программу, реализующую метод бинарного дерева для построения таблицы идентификаторов. Для организации дерева использовать динамический массив. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами.

3. Написать программу, реализующую создание таблицы идентификаторов на основе метода рехэширования. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами. В качестве хэш-функции использовать:

а) коды первых двух букв идентификаторов;

б) коды последних двух букв идентификаторов;

в) некоторое случайное число в диапазоне от -10 до 10;

г) номер текущего вычисления хэш-функции.

4. Написать программу, реализующую создание таблицы идентификаторов на основе метода цепочек. В качестве хэш-функции использовать варианты из предыдущего задания.

5. Сравнить реализованные методы построения таблиц идентификаторов.

Конечные автоматы

Распознавателем языка называется программа, которая, получая на вход цепочку символов входного алфавита, принимает ее, если она представляет собой предложение языка и не принимает иначе.

Теория автоматов лежит в основе теории построения компиляторов. Под автоматом понимают не реально существующее устройство, а некоторую математическую модель, свойства и поведение которой можно изучать.

Конечный автомат является простейшим из моделей теории автоматов и служит управляющим устройством для всех остальных автоматов.

Преимуществами конечного автомата являются следующие факты:

моделирование конечного автомата требует фиксированного объема памяти;

существует ряд теорем и алгоритмов, позволяющих конструировать и упрощать конечные автоматы;

обработка одного входного символа требует небольшого числа операций, что обеспечивает быстроту работы.

Конечный автомат может решать простые задачи компиляции (в частности, лексический блок почти всегда строится на его основе).

На вход конечного автомата подается цепочка символов из конечного множества, называемого входным алфавитом. Как допускаемые, так и отвергаемые автоматом цепочки состоят из символов входного алфавита. Символы, не принадлежащие входному алфавиту, нельзя подавать на вход автомата.

Работа конечного автомата представляет последовательность шагов (тактов). На каждом шаге автомат находится в одном из своих состояний. Одно из этих состояний называется начальным. На каждом следующем шаге автомат может либо остаться в текущем состоянии, либо перейти в другое. То, в какое состояние перейдет автомат, определяет функция переходов. Она зависит не только от текущего состояния, но и от символа на входе автомата. Если функция переходов допускает несколько состояний, то автомат может перейти в любое из них.

Некоторые состояния выбираются в качестве допускающих (или заключительных) состояний. Если автомат, начав работу в начальном состоянии, при прочтении всей входной цепочки переходит в одно из допускающих состояний, говорят, что входная цепочка допускается автоматом. Если последнее состояние автомата не является допускающим – цепочка отвергнута.

Таким образом, конечный автомат задается пятеркой вида $M(Q, V, \delta, q_0, F)$, где

Q – конечное множество состояний автомата,

V – алфавит входных символов,

δ – функция переходов, $\delta(a,q) = R$, $a \in V, q \in Q, \quad R \subseteq Q$,

q_0 – начальное состояние автомата ($q_0 \in Q$),

F – непустое множество конечных состояний автомата.

Конфигурация автомата на каждом шаге работы определяется тройкой (q, ω, n), где q – текущее состояние автомата, ω – цепочка входных символов, n – положение указателя во входной цепочке. Тогда начальная конфигурация автомата $(q_0, \omega, 0)$.

Языком автомата называют множество всех цепочек, которые принимаются этим автоматом. Два конечных автомата эквивалентны, если они задают один и тот же язык.

Конечный автомат часто представляют в виде диаграммы или графа переходов автоматов. Граф переходов конечного автомата – это направленный граф, вершины которого помечены символами состояний конечного автомата, а дуга, помечена некоторым символом, если определена соответствующая функция перехода δ. Начальное и конечное состояние автомата помечаются специальным образом.

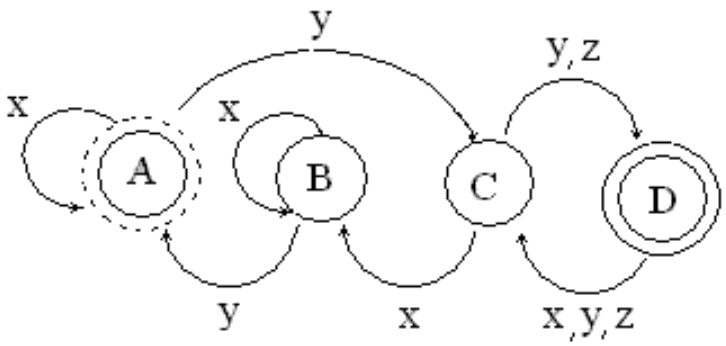


Рис. 5. Граф перехода конечного автомата

На рис. 5 задан конечный автомат *M*({ *A, B, C, D*}, {*x, y, z*}, *δ, A,*, {*D*}*);*
δ: δ(A, x)=A, δ(A, y)=C), δ(B, x)=B, δ(B, y)=A, δ(C, x)=B, δ(C, y)=D, δ(C, z)=D, δ(D, x)=C, δ(D, y)=C, δ(D, z)=C

Конечный автомат называют полностью определенным, если в каждом его состоянии существует функция перехода для всех возможных входных символов.

Для моделирования работы конечного автомата его удобно привести к полностью определенному виду, чтобы исключить ситуации, из которых нет переходов по входным символам. Для этого добавляют еще одно состояние, которое условно называют «ошибка» – E. На него замыкают все неопределенные переходы, в том числе и само на себя.

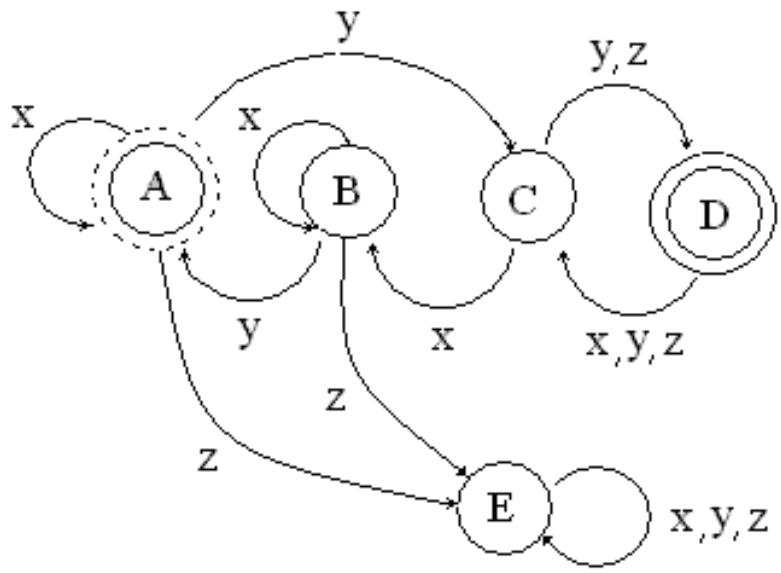


Рис. 6. Граф переходов полностью определенного конечного автомата

Другой способ представления конечного автомата – таблица переходов, в которой информация размещается в соответствии со следующими соглашениями:

столбцы помечены входными символами,

строки помечены символами состояний,

элементами таблицы являются символы новых состояний, соответствующих входным символам столбцов и состояниям строк,

первая строка помечена символом начального состояния,

строки, соответствующие допускающим (заключительным) состояниям помечены справа единицами, а строки, соответствующие отвергающим состояниям, помечены нулями.

Таблица переходов полностью определенного конечного автомата, представленного на рис. 6 задается таблицей:

	x	*y*	*z*	
A	A	C	E	0
B	B	A	E	0
C	B	D	D	1
D	C	C	C	0


E
E
E
E
0

Конечный автомат называют детерминированным конечным автоматом, если в каждом из его состояний для любого входного символа функция перехода содержит не более одного состояния. В противном случае автомат называют недетерминированным. Доказано, что для любого конечного автомата можно построить эквивалентный ему детерминированный конечный автомат. Моделировать работу детерминированного конечного автомата существенно проще, поэтому всегда стремятся сделать это преобразование. При построении компиляторов чаще всего используют полностью определенный детерминированный конечный автомат.

Конечные автоматы являются распознавателями для регулярных языков. Поскольку язык констант и идентификаторов является регулярным, то для реализации лексических анализаторов служат регулярные грамматики и конечные автоматы.

Среди всех регулярных грамматик выделяют отдельный класс – автоматные грамматики. Они также могут быть леволинейными и праволинейными.
Разница между автоматными и обычными регулярными грамматиками заключается в том, что где в правилах обычных регулярных грамматик может присутствовать цепочка терминальных символов, в автоматных грамматиках может присутствовать только один терминальный символ. Любая автоматная грамматика является регулярной, но не наоборот – не всякая регулярная грамматика является автоматной.

Доказано, что классы обычных регулярных грамматик и автоматных грамматик почти эквивалентны.

Существует алгоритм, который позволяет преобразовать произвольную регулярную грамматику к автоматному виду – то есть построить эквивалентную ей автоматную грамматику:

Шаг 1. Все нетерминальные символы из множества **VN** исходной грамматики *G* переносятся во множество **VN'** грамматики G'.

Шаг 2. Необходимо просматривать все множество правил P грамматики G. Если встречаются правила вида $A \rightarrow Ba_1$, $A, B \in$ **VN**, $a_1 \in$ **VT** или вида $A \rightarrow a_1$, $A \in$ **VN**, $a \in$ **VT**, то они переносятся во множество P' правил грамматики G^1 без изменений.

Если встречаются правила вида $A \rightarrow Ba_1, a_2, ..., a_n$, $n > 1$, $A, B \in$ **VN**, $\forall\ n > i > 0$: $a_i \in$ **VT**, то во множество нетерминальных символов **VN'** грамматики G' добавляются символы $A_1, A_2, ..., A_{n-1}$, а во множество правил P' грамматики G' добавляются правила:

$A \rightarrow A_{n-1} a_n$

$A_{n-1} \rightarrow A_{n-2} a_{n-1}$

...

$A_2 \rightarrow A_1 a_2$

$A_1 \rightarrow Ba_1$

Если встречаются правила вида $A \rightarrow a_1 a_2, ..., a_n$, $n > 1$, $A, B \in$ **VN**, $\forall\ n > I > 0$: $a_i \in$ **VT**, то во множество нетерминальных символов **VN'** грамматики G' добавляются символы $A_1, A_2, ..., A_{n-1}$, а во множество правил P′ грамматики G' добавляются правила:

$A \rightarrow A_{n-1} a_n$

$A_{n-1} \rightarrow A_{n-2} a_{n-1}$

...

$A_2 \rightarrow A_1 a_2$

$A_1 \rightarrow a_1$

Если встречаются правила вида $A \rightarrow B$ или вида $A \rightarrow \lambda$, то они переносятся во множество правил P' грамматики G' без изменений.

Шаг 3. Просматривается множество правил P' грамматики G' с целью поиска правил вида $A \rightarrow B$ или вида $A \rightarrow \lambda$.

Если находится правило первого вида, то просматривается множество правил P’ грамматики G'. Если в нем присутствуют правила вида $B \to C$, $B \to Ca$, $B \to a$ или $B \to \lambda$, то в него добавляются правила вида $A \to C$, $A \to Ca$, $A \to a$ и $A \to \lambda$ соответственно, $\forall A, B, C \in$ **VN'**, $\forall a \in$ **VT'** (при этом учитывается, что в грамматике не должно быть совпадающих правил). Правило $A \to B$ удаляется из множества правил P'.

Если находится правило вида $A \to \lambda$ (и символ A не является целевым символом S), то просматривается множество правил P' грамматики G'. Если в нем присутствуют правила вида $B \to A$ или $B \to Aa$, то в него добавляются правил: вида $B \to \lambda$ и $B \to a$ соответственно, $\forall A, B, C \in$ **VN'**, $\forall a \in$ **VT'** (при этом также учитывается, что в грамматике не должно быть совпадающих правил). Правило $A \to \lambda$ удаляется из множества правил P'.

Шаг 4. Если на шаге 3 было найдено хотя бы одно правило вида $A \to B$ или $A \to \lambda$ во множестве правил P' грамматики G', то надо повторить шаг 3, иначе перейти к шагу 5.

Шаг 5. Целевым символом S' грамматики G' становится символ S.

Шаги 3 и 4 алгоритма можно не выполнять, если грамматика не содержит правил вида A → B (такие правила называются цепными) или вида A → λ (такие правила называются λ - правилами). Реальные регулярные грамматики обычно не содержат правил такого вида.

В алгоритме рассмотрен случай леволинейной грамматики. Для праволинейной легко построить аналогичный алгоритм.

Регулярные языки являются удобным типом языков. Для них разрешимы многие проблемы: эквивалентности двух языков, принадлежности языку заданной цепочки символов, пустоты языка.

Эти проблемы разрешимы вне зависимости от того, каким из трех способов задан язык:

1) регулярными грамматиками (праволинейным или леволинейными);
2) конечным автоматом;
3) регулярным множеством.

Все три способа равноправны. Существуют алгоритмы, которые позволяют для регулярного языка, заданного одним из способов, построить другой способ задания того же самого языка.

Регулярные множества – это множества цепочек символов над заданным алфавитом, построенные с использованием операций объединения, конкатенации и итерации.

Контрольные вопросы

1. Может ли граф переходов конечного автомата использоваться для однозначного определения автомата? Почему?
2. От каких параметров зависит функция переходов конечного автомата?
3. В каком случае конечный автомат называется полностью определенным?
4. Всегда ли недетерминированный конечный автомат может быть приведен к детерминированному?
5. Сколькими параметрами определяется конфигурация конечного автомата?
6. Распознавателями какого типа языков являются конечные автоматы?

Задание

1. Построить конечный автомат для следующих видов цепочек, состоящих из нулей и единиц:

а) между вхождениями единиц четное число нулей;

б) за каждым вхождением пары единиц следует нуль;

в) каждый пятый символ единица;

г) все цепочки начинаются на нуль и оканчиваются единицей;

д) в цепочке перед каждой единицей стоит нуль;

е) цепочка должна содержать ровно три единицы.

Для реализации конечного состояния построить граф или таблицу переходов, составить программу на языке высокого уровня.

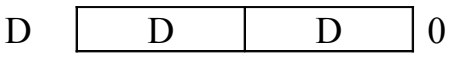
2. Построить конечный автомат, распознающий зарезервированные слова языка C++: inline, function, class, protected, extern, delete, operator, struct.

3. Построить регулярное выражение для представления десятичных чисел, описания переменных в алгоритмических языках.

4. Описать словами множество цепочек, распознаваемых каждым из конечных автоматов, заданных таблицами переходов:

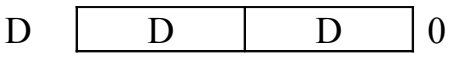
а)

	0	*1*	
A	B	C	0
B	D	B	1
C	C	D	1


D
D
D
0

б)

	0	*1*	
A	B	A	0
B	D	C	0
C	C	D	1


D
D
D
0

в)

	x	*y*	*z*	
A	A	C	C	0
B	C	D	C	0
C	C	C	C	0

D	C	C	A	1

Преобразования конечного автомата

Алгоритм преобразования произвольного конечного автомата $M(Q, V, \delta, q_0, F)$ к эквивалентному ему, детерминированному конечному автомату $M'(Q', V, \delta', q_0', F')$, заключается в следующем:

Шаг 1. Множество состояний Q' автомата M' строится комбинацией всех состояний множества Q автомата M. Их возможное число 2^n-1, где n – количество состояний.

Шаг 2. Функция переходов δ' автомата M' строится как $\delta'(a, [q_1, q_2, ..., q_m]) = [r_1, r_2, ..., r_k]$, где $\forall\ 0< i \le m\ \exists\ 0< j \le k$ такое, что $\delta(a, q_i) = r_j$.

Шаг 3. Обозначим $q'_0 = [q_0]$.

Шаг 4. Если $f_1, f_2, ..., f_l$ $(l > 0)$ – конечные состояния автомата M ($f_i \in F$), тогда множество конечных состояний F' автомата M' строится из всех состояний имеющих вид […, f_i, …].

Затем требуется из полученного автомата удалить недостижимые символы по следующему алгоритму:

Шаг 1. Обозначим множество достижимых состояний R, $R=\{q_0\}$, а множество текущих активных состояний на каждом шаге алгоритма P_i. $i=0$, $P_0=\{q_0\}$.

Шаг 2. $P_{i+1}=\varnothing$.

Шаг 3. $\forall\ a \in V$, $\forall\ q \in P_i$ $P_{i+1}=P_i \cup \delta(a, q)$

Шаг 4. Если $P_{i+1} - R = \varnothing$ алгоритм завершен, иначе $R=R \cup P_{i+1}$, $i = i + 1$, перейти к шагу 3.

После этого можно исключить все состояния, не вошедшие во множество R.

Пример.

Преобразовать конечный автомат M ({H, A, B, S}, {x, y}, δ, H, {S}), δ(H, y)=B, δ(B, x)=A, δ(A, y)={B, S}. Граф переходов такого автомата изображен на рис. 7.

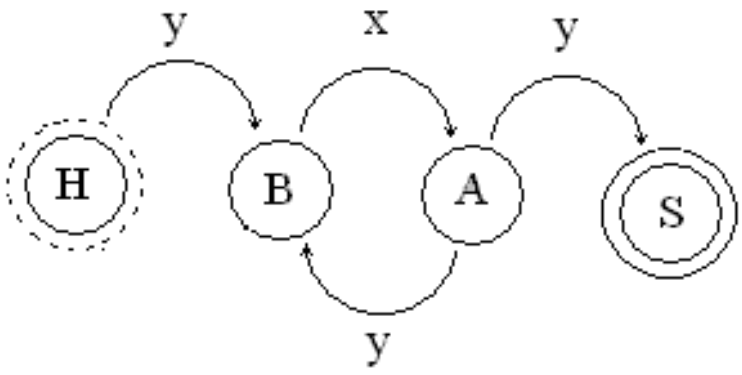


Рис. 7. Граф переходов недетерминированного конечного автомата

Данный автомат недетерминированный, поскольку из состояния A возможны два различных перехода по символу y.

Шаг 1. Построим множество состояний эквивалентного автомата Q′ = {[H], [A], [B], [S], [HA], [HB], [HS], [AB], [AS], [BS], [HAB], [HAS], [HBS], [ABS], [HABS]}.

Шаг 2. Функция переходов эквивалентного конечного автомата

δ′ ([H], y)=[B]

δ′ ([A], y)=[BS]

δ′ ([B], x)=[A]

δ′ ([HA], y)=[BS]

δ′ ([HB], x)=[A]

δ′ ([HS], y)=[B]

δ′ ([AB], x)=[A]

δ′([AB], y)=[BS]

δ′ ([AS], y)=[BS]

δ′ ([BS], x)=[A]

δ′ ([HAB], x)=[A]

δ′ ([HAS], y)=[BS]

δ′ ([HBS], y)=[B]

δ′([HBS], x)=[A]

δ′ ([ABS], y)=[BS]

δ′ ([ABS], x)=[A]

δ′ ([HABS], x)=[A]

δ′ ([HABS], y)=[BS]

Шаг 3. Начальное состояние M′ q'_0=[H]

Шаг 4. Множество конечных состояний эквивалентного детерминированного конечного состояния F′ = {[S], [HS], [AS], [BS], [HAS], [HBS], [ABS], [HABS]}

Исключим недостижимые состояния, в итоге получаем

M′({H, B, A, S}, {x, y}, δ′, H, {S}), δ′(H, y)=B, δ′(B, x)=A, δ′(A, y)=S, δ′(S, x)=A.

Граф переходов этого автомата приведен на рис.8.

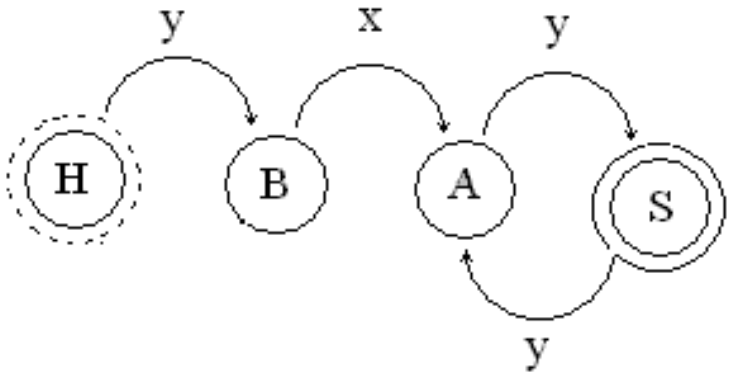


Рис. 8. Граф переходов детерминированного конечного автомата

Моделировать работу детерминированного конечного автомата существенно проще, чем произвольного конечного автомата. Но при выполнении преобразований число состояний автомата может значительно вырасти. Тогда затраты на моделирование возрастают, и преобразование не оправдывается.

Многие конечные автоматы можно минимизировать. Минимизация конечного автомата заключается в построении эквивалентного конечного автомата с меньшим числом состояний.

Для минимизации автомата используется алгоритм построения эквивалентных состояний конечного автомата. Два различных состояния q и q’ в конечном автомате M(Q, V, δ, q_0, F) называются n-эквивалентными (n-неразличимыми) $n \geq 0$, если, находясь в одном из этих состояний и получив на вход любую цепочку символов, автомат может перейти в одно и то же множество конечных состояний. Очевидно, что 0-эквивалентными состояниями автомата M(Q, V, δ, q_0, F) являются два множества его состояний F и F–Q.

Множества эквивалентных состояний автомата называют классами эквивалентности, а всю совокупность – множеством классов эквивалентности R(n), причем R(0) = {F , F–Q}.

Алгоритм минимизации конечного автомата заключается в следующем:

Шаг 1. Из автомата исключаются все недостижимые состояния.

Шаг 2. Строятся классы эквивалентности автомата.

Шаг 3. Классы эквивалентности состояний исходного конечного автомата становятся состояниями результирующего минимизированного конечного автомата.

Шаг 4. Функции переходов результирующего конечного автомата очевидным образом строится на основе функции переходов исходного конечного автомата.

Для этого алгоритма доказано: во-первых, что он строит минимизированный конечный автомат, эквивалентный заданному конечному автомату; во-вторых, что он строит конечный автомат с минимально возможным числом состояний (минимальный конечный автомат).

Если два состояния q_1 и q_2 одного автомата эквивалентны, то автомат можно упростить, заменяя в графе переходов (таблице переходов) все вхождения имен этих состояний каким-нибудь новым именем, а затем, удаляя двух строк, соответствующих q_1 и q_2. Например, состояния K и L конечного автомата, заданного таблицей переходов, представленной на рис. 9, явно имеют одинаковые функции, так как оба являются допускающими, оба переходят в состояние B при чтении входного символа *a* и оба переходят в состояние C при чтении *b*.

	a	*b*	
A	A	K	0
B	C	L	1
C	L	A	0
K	B	C	1
L	B	C	1

Рис. 9. Таблица переходов не минимизированного автомата

Поэтому можно объединить состояния K и L в одно состояние и назвать его X. Получается упрощенная таблица состояний, представленная на рис.10.

Обычно эквивалентность состояний менее очевидна. Два состояния эквивалентны тогда и только тогда, когда не существует различающей их цепочки.

	a	*b*	
A	A	4	0
B	C	X	1
C	X	A	0
X	B	C	1

Рис.10. Таблица переходов минимизированного автомата

Метод проверки эквивалентности состояний основывается на следующем. Состояния q_1 и q_2 эквивалентны тогда и только тогда, когда выполняются два условия:

1) условие подобия – состояния q_1 и q_2 должны быть либо оба допускающие, либо оба отвергающие;

2) условие преемственности – для всех входных символов состояния q_1 и q_2 должны переходить в эквивалентные состояния, т.е. их преемники эквивалентны.

Эти условия выполняются тогда и только тогда, когда q_1 и q_2 не имеют различающей цепочки. Если нарушено одно из условий, существует цепочка, различающая эти два состояния. А если не выполняется условие подобия, различающей является пустая цепочка. Если нарушено условие преемственности, то неко-

торый входной символ x переводит из состояния q_1 и q_2 в неэквивалентные. Поэтому x с приписанной к нему цепочкой, различающей эти новые состояния, образует цепочку, различающую q_1 и q_2.

Рассмотренные условия можно использовать в общем методе проверки на эквивалентность произвольной пары состояний. Для этого строятся таблицы эквивалентности состояний. Рассмотрим конечный автомат, таблица переходов которого изображена на рис. 11. Выявим его эквивалентные состояния.

Сначала проверяем на эквивалентность состояния F и P. Таблица эквивалентности состояний содержит по одному столбцу для каждого входного символа, для *y* и *z*. Следующие строки будут добавляться в ходе проверки. Первоначально такая таблица имеет вид, представленный на рис.12: имеется одна строка, которая помечена парой состояний, подвергаемых проверке, т.е. F, P. Условие подобия для этих строк выполняется, так как оба состояния являются отвергающими.

	y	*z*	
F	F	K	0
G	H	M	0
H	H	P	0
K	N	P	0
L	G	N	1
M	N	M	0
N	N	K	1
P	N	K	0

Рис. 11. Таблица переходов конечного автомата до минимизации

	y	*z*
F, P		

Рис.12. Первоначальный вид таблицы эквивалентности состояний

Для проверки условия преемственности результат действия на отдельную пару состояний каждого входного символа запишем в соответствующую ячейку таблицы эквивалентности. Так как состояние F, P под действием входного символа *y* переходит в состояние F и N соответственно, то они записываются в столбец таблицы, соответствующий символу *y*. Так как оба состояния F и P переводятся

символом *z* в состояние K, соответственно K помещается в столбец для z. Таблица эквивалентности символов F, P примет вид, представленный на рис.13.

	y	*z*
F, P	F, N	K

Рис. 13. Строка таблицы эквивалентности состояний F, P

Чтобы нарушалось условие преемственности, должны быть неэквивалентны либо состояние F и N, либо состояния K и K. Так как каждое состояние эквивалентно само себе, состояния K и K эквивалентны автоматически.

Для исследования на эквивалентность состояния F и N, к таблице эквивалентности состояний добавляется новая строка, помеченная новой парой F, N. Для нее повторяется весь процесс, описанный для пары F, P. Условие подобия для этой пары не выполняется, так как N – допускающее, а F – отвергающее состояние. Следовательно, состояния F и N неэквивалентны. Таблицу эквивалентности состояний можно использовать для построения различающей цепочки. Строка F, N появилась как результат применения входного символа y к паре F, P, поэтому *y* является различающей цепочкой.

Строим таблицу эквивалентности пары F, G. Эти состояния подобны, поэтому требуется вычислить результат применения к ним каждого входного символа, полученные состояния размещаются в таблице. Надежды на эквивалентность пары F, G оправдаются, если будет установлена эквивалентность пар, помещенных в таблицу – F, H и K, M. В таблицу добавляется строка для каждой из этих пар (см. рис. 14).

	y	*z*
F, G	F, H	K, M

	y	*z*
F, G	F, H	K, M
F, H		
K, M		

Рис. 14. Таблица эквивалентности состояний F, G

Результаты промежуточного заполнения таблицы представлены на рис.15.

	y	*z*
F, G	F, H	K, M
F, H	F, H	K, P
K, M		

	y	*z*
F, G	F, H	K, M
F, H	F, H	K, P
K, M	N	M, P
K, P		
M, P		

Рис. 15. Заполнение таблица эквивалентности состояний F, G

Один из двух новых элементов дает новую строку, а именно пара K, P. Другой элемент уже имеется в таблице – проверять его не следует. Далее осуществляется проверка следующей по списку пары: K, M. Состояния этой пары подобны. Вычисляем пары N, N и M, P. Так как состояние N эквивалентно самому себе, единственной новой строкой в таблице будет M, P.

Описанные процедуры повторяются для оставшихся пар – K, P и M, P. По их окончании не удается получить ни одной новой пары неподобных состояний и ни одной пары, которую надо проверять на подобие. Окончательная таблица представлена на рис. 16.

	y	*z*
F, G	F, H	K, M
F, H	F, H	K, P
K, M	N	M, P
K, P	N	K, P
M, P	N	K, M

Рис.16. Итоговая таблица эквивалентности состояний F, G

Различающие цепочки для состояний F, G отсутствуют, поэтому эти состояния являются эквивалентными.

Алгоритм проверки эквивалентности двух состояний можно описать следующим образом.

Шаг 1. Начать построение таблицы эквивалентности состояний с отведения столбца для каждого входного символа. Пометить первую строку парой проверяемых состояний.

Шаг 2. Выбрать в таблице эквивалентности состояний строку, ячейки которой еще не заполнены, и проверить, подобны ли состояния, которыми она помечена. Если они не подобны, то два исходных состояния неэквивалентны. Перейти к шагу 3. Иначе вычислить результат применения каждого входного символа к этой паре состояний и записать полученные пары состояний в соответствующие ячейки рассматриваемой строки.

Шаг 3. Если элементом таблицы является пара одинаковых состояний или пара состояний, которые уже использовались как метки строк – дополнительные действия не требуются. Если же элемент таблицы – пара различных состояний, до этого не используемая как метка, добавляется новая строка. Порядок состояний в паре не важен, и пары q_1, q_2 и q_2, q_1 считаются одинаковыми.

Шаг 4. Если все строки таблицы эквивалентности заполнены, исходная пара состояний и все пары, порожденные в ходе проверки, эквивалентны, проверка закончена. Если же таблица не заполнена, нужно обработать еще, по крайней мере, одну ее строку и применить шаг 2.

Так как каждая пара, появившаяся в заполненной таблице, содержит эквивалентные состояния, этот метод проверки дает обычно больше информации, чем предполагалось сначала. Из итоговой таблицы эквивалентности (рис. 16) следует, что, кроме эквивалентности пары (F, G), которая подвергалась проверке, доказана эквивалентность пар (F, H), (K, M), (K, P), (M, P). По свойству транзитивности из эквивалентности пар состояний F, H и F, G и следует эквивалентность пары G, H. Таким образом, состояния F, G, H эквивалентны друг другу. Аналогично, эквивалентны друг другу состояния K, M, P.

Автомат можно упростить, объединив состояния F, G, H в состояние A, а K, M, P – в состояние B. Новые имена подставляются в таблицу переходов, лиш-

ние строки удаляются и получается более простой – эквивалентный, изображенному на рис. 11, новый автомат представленный таблицей переходов на рис. 17.

	y	*z*	
A	A	B	0
B	N	B	0
L	A	N	1
N	N	B	1

Рис. 17. Таблица переходов минимального конечного автомата

Чтобы упростить автомат необходимо, также удалить из него состояния, недостижимые из начального состояния ни для какой входной цепочки.

Минимизация конечных автоматов позволяет уменьшить количество их состояний, что в дальнейшем упрощает функционирование распознавателя.

Контрольные вопросы

1. В чем заключается алгоритм преобразования конечного автомата к детерминированному виду?
2. В каких случаях преобразовывать конечный автомат к детерминированному виду нецелесообразно?
3. Какие два состояния автомата называются эквивалентными?
4. Что собой представляет таблица эквивалентных состояний? Каким образом она заполняется?
5. Как определяется недостижимое состояние?
6. Какое из преобразований приводит к уменьшению количества состояний конечного автомата, а какое к их уменьшению?

Задание

1. Найти различающую цепочку для пары автоматов:

	a	*b*	
A	A	B	1
B	C	D	0
C	D	A	1
D	A	B	0

	a	*b*	
A	A	D	1
B	A	D	0
C	B	A	1
D	C	B	0

	0	*1*	
S1	S1	S3	0
S2	S7	S4	1
S3	S6	S5	0
S4	S1	S4	1
S5	S1	S4	0
S6	S7	S6	1
S7	S7	S3	0

	0	*1*	2	
A	C	E	G	0
B	J	E	G	1
C	J	A	H	0
D	F	A	G	1
E	E	J	H	0
F	D	I	A	1
G	H	A	J	1
H	G	J	B	2
I	D	F	G	3
J	B	H	G	4
				5
				6
				7

X	*Y*	
4	1	1
5	1	1
4	5	0
2	6	0
1	7	0
1	4	1
2	5	1

2. Найти минимальную эквивалентную таблицу для каждого из ниже расположенных автоматов.

3. Для автоматов из предыдущего задания найти недостижимые состояния.

4. Найти недостижимые состояния автомата, представленного таблицей состояний

Автоматы с магазинной памятью

Синтаксический анализатор (синтаксический разбор) – часть компилятора, которая отвечает за выявление и проверку синтаксических конструкций входного языка. В задачи синтаксического анализа входят:

поиск и выделение синтаксических конструкций в тексте исходной программы;

установка типа и проверка правильности каждой из найденных синтаксических конструкций;

представление синтаксических конструкций в виде, удобном для дальнейшей генерации текста результирующей программы.

Синтаксический анализатор является основной частью компилятора на этапе анализа, поскольку без него работа компилятора бессмысленна. В то время как лексический разбор является необязательной фазой. Синтаксический анализатор воспринимает выход лексического анализатора, если он имеется в наличие, либо сам распознает их. На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие – на синтаксическом. Обычно это определяется разработчиком компилятора, исходя из технологических аспектов программирования, а также синтаксиса и семантики входного языка.

В основе синтаксического анализатора лежит распознаватель текста программы, построенный на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью контекстно-свободных грамматик, реже регулярных грамматик.

Чаще всего регулярные грамматики применимы к языкам ассемблера, а синтаксис языков высокого уровня основан на грамматике КС-языков.

Распознавателями для КС-языков являются автоматы с магазинной памятью (МП-автоматы). Это односторонние недетерминированные распознаватели с линейно ограниченной магазинной памятью.

В общем виде МП-автомат можно определить как R (Q, V, Z, δ , qo, Zo, F), где Q – множество состояний автомата;

V – алфавит входных символов автомата;

Z – специальный конечный алфавит магазинных символов автомата;

δ – функция переходов автомата;

$q_o \in Q$ – начальное состояние автомата;

$z_o \in Z$ – начальный символ магазина;

$F \in Q$ – множество конечных состояний.

МП-автомат в отличие от обычного КА имеет стек (магазин), в который можно помещать специальные (магазинные) символы. Обычно это терминальные и нетерминальные символы грамматики языка. Переход МП-автомата из одного состояния в другое зависит не только от входного символа, но и от символа на верхушке стека. Таким образом, конфигурация автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки (положением указателя в цепочке) и содержимым стека.

МП-автомат условно можно представить в виде схемы, показанной на рис. 18.

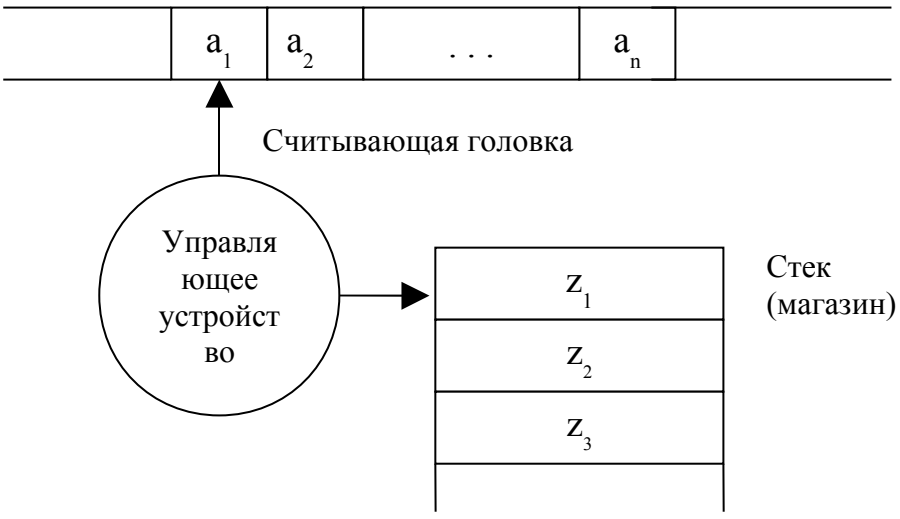


Рис. 18. Схема работы МП-автомата

Каждый шаг процесса обработки задается множеством правил, использующих информацию трех видов: состояние, верхний символ магазина, текущий входной символ.

Это множество правил называется управляющим устройством, или механизмом управления.

В зависимости от получаемой информации управляющее устройство выбирает либо выход из процесса (т. е. прекращает обработку), либо переход в новое состояние. Переход состоит из трех операций: над магазином, над состоянием и над входом. Возможные операции над магазином могут быть следующими:

1) втолкнуть в магазин определенный магазинный символ;

2) вытолкнуть верхний символ магазина;

3) оставить магазин без изменений.

Операция над состоянием единственна – перейти в заданное новое состояние.

Возможные операции над входом:

1) перейти к следующему входному символу и сделать его текущим входным символом;

2) оставить данный входной символ текущим, иначе говоря, держать его до следующего шага.

Обработку входной цепочки МП-автомат начинает в некотором выделенном состоянии при определенном содержимом магазина, а текущим входным символом является первый символ входной цепочки. Затем автомат выполняет операции, задаваемые его управляющим устройством. Если происходит выход из процесса, обработка прекращается; если происходит переход, то он дает новый верхний магазинный символ, новый текущий символ – автомат переходит в новое состояние, и управляющее устройство определяет новое действие, которое нужно произвести.

Чтобы управляющие правила имели смысл, автомат не должен требовать следующего входного символа, если текущим символом является концевой маркер, и не должен выталкивать символ из магазина, если это маркер дна. Поскольку маркер дна может находиться исключительно на дне магазина, автомат не должен также вталкивать его в магазин.

При выполнении такта (перехода) из стека удаляется верхний символ, соответствующий условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится верхушкой стека.

МП-автомат допускает (принимает) цепочку символов, если, получив эту цепочку ход, он может перейти в одну из конечных конфигураций – когда при окончании цепочки автомат находится в одном из конечных состояний, а стек содержит некоторую определенную цепочку. Иначе цепочка символов не принимается.

МП-автомат допускает цепочку символов с опустошением магазина, если при окончании разбора цепочки автомат находится в одном из конечных состояний, а стек пуст.

Кроме обычного МП-автомата существует понятия расширенного и детерминированного МП-автомата.

Расширенный МП-автомат может заменять цепочку символов конечной длины в верхней части стека на другую цепочку символов конечной длины.
В отличие от обычного МП-автомата, который на каждом такте работы может изымать из стека только один символ, расширенный МП-автомат может изымать за один такт цепочку символов, находящуюся на вершине стека.

МП-автомат называется *детерминированным,* если из каждой его конфигурации возможно не более одного перехода в следующую конфигурацию. В противном случае МП-автомат называется недетерминированным

Стандартным представлением МП-автомата с одним состоянием является таблица со столбцами для входных символов и строками для символов магазина (рис. 19).

( ) ┤

	A		
Магазинный символ	ВТОЛКНУТЬ (А) СДВИГ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ
	ВТОЛКНУТЬ (А) СДВИГ	ОТВЕРГНУТЬ	ДОПУСТИТЬ

Рис. 19. Таблица состояний МП-автомата

Контрольные вопросы

1. Каковы задачи, решаемые на этапе синтаксического анализа?

2. Если из каждой конфигурации автомата с магазинной памятью возможно не более одного перехода в следующую конфигурацию, то он называется

а) расширенным б) детерминированным

в) недетерминированным г) обычным

3. Автоматы с магазинной памятью служат для распознавания цепочек языков на основе

а) контекстно-свободной грамматики

б) грамматики с фразовой структурой

в) контексно-зависимой грамматики

г) регулярной грамматики

4. Каковы отличия автомата с магазинной памятью от конечного автомата? Какой из них является более интеллектуальным?

5. Объяснить понятие «управляющее устройство».

6. В каких случаях входная цепочка допускается МП-автоматом?

Задание

1. Построить МП-распознаватель для каждого из следующих множеств цепочек:

a) $\{1^n\ 0^m \mid n > m > 0\}$;

b) $\{1^n\ 0^m \mid n \geq m > 0\}$;

c) $\{1^n\ 0^n\ 1^m\ 0^m \mid n, m \geq 0\}$;

d) $\{1^n\ 0^m\ 1^n\ 0^m \mid n, m \geq 0\}$;

e) $\{1^n\ 0^m \mid m > n > 0\}$.

2. Для каждого из множеств первого задания указать цепочку длины, большей трех. Показать последовательность конфигураций соответствующих автоматов, построенных в первом задании при распознавании каждой из цепочек.

3. Написать три цепочки, принадлежащие множеству, распознаваемому МП-автоматом с одним состоянием, представленным следующей таблицей:

	a	b	c	┤
A	ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ
B	ВТОЛКНУТЬ (C) СДВИГ	ВЫТОЛКНУТЬ ДЕРЖАТЬ	ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ
C	ВТОЛКНУТЬ (B) ДЕРЖАТЬ	ВТОЛКНУТЬ (C) СДВИГ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ
Магазинный символ	ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ	ОТВЕРГНУТЬ	ДОПУСТИТЬ

Для каждой из этих цепочек указать соответствующие последовательности конфигураций, допускающие эти цепочки.

4. Привести пример такого множество цепочек, которое может распознать МП-автомат с магазинным алфавитом, содержащим маркер дна магазина и еще два символа, но не может распознать никакой МП-автомат, у которого магазинный алфавит состоит из маркера дна магазина и еще одного символа.

5. Составить три допускаемые цепочки и соответствующие последовательности конфигураций для МП-автомата с одним состоянием, представленного на рис.19.

Преобразование контекстно-свободных грамматик

Процесс построения синтаксического анализатора значительно сложнее процесса создания лексического анализатора. Это обусловлено тем, что контекстно-свободные грамматики и МП-автоматы сложнее, чем регулярные грамматики и конечные автоматы.

Преобразование КС-грамматик преследует две основные цели: упрощение правил грамматики и облегчение создания распознавателя языка. Не всегда эти цели возможно совместить. При создании компилятора для языков программирования вторая цель является основной.

Все преобразования КС-грамматик делятся на две группы:

1) преобразования, связанные с исключением из грамматики избыточных правил и символов, без которых она может существовать;

2) преобразования, в результате которых изменяется вид и состав правил, при этом грамматика может дополняться новыми правилами и новыми нетерминальными символами.

В результате преобразований всегда получается новая КС-грамматика, эквивалентная исходной, т.е. определяющая тот же язык.

Приведенной называют КС-грамматику, не содержащую недостижимые и бесплодные символы, циклы (цепные правила) и правила с пустыми цепочками. Шаги преобразования грамматики к приведенному типу должны строго выполняться в порядке указанном ниже.

1)Удаление бесплодных правил

Символ $A \in VN$ называется *бесплодным* тогда и только тогда, когда из него нельзя вывести ни одной цепочки терминальных символов. В простейшем случае символ бесплодный, если во всех правилах, где он стоит в левой части, он встречается и в правой. Иначе предполагаются зависимости между цепочками бесплодных символов, которые в любой последовательности вывода порождают друг друга.

Алгоритм удаления бесплодных правил предполагает создание специального множества нетерминальных символов. Обозначим его *Y*. Первоначально оно содержит только те символы, из которых можно непосредственно вывести терминальные цепочки. Затем оно пополняется на основе правил исходной грамматики.

Шаг 1. Y_0 = пустое множество, *i*=0.

Шаг 2. $Y_i = \{A \mid (A \to \alpha) \in P, \alpha \in (Y_{i-1} \cup VT)^*\} \cup Y_{i-1}$

Шаг 3. Если $Y_i \neq Y_{i-1}$, то $i = i+1$ и перейти к шагу 2, иначе перейти к шагу 4.

Шаг 4. $VN' = Y_i, VT' = VT$ во множество правил P' входят только те правила из P, которые содержат только символы из множества $(VT \cup Y_i), S' = S$.

2) Удаление недостижимых символов

Символ называется *недостижимым*, если он не участвует ни в одной цепочке вывода из целевого символа грамматики.

Первоначально во множество достижимых символов *V* входит только целевой символ, а затем оно пополняется на основе правил грамматики.

Шаг 1. $V_0 = \{S\}$, *i*=0.

Шаг 2. $V_i = \{x \mid x \in (VT \cup VN) u (A \to \alpha x \beta) \in P, A \in V_{i-1}, \alpha, \beta \in (VT \cup VN)^*\} \cup V_{i-1}$.

Шаг 3. Если $V_i \neq V_{i-1}$, то *i*=*i*+1 и перейти к шагу 2, иначе перейти к шагу 4.

Шаг 4. $VN' = VN \cap V_i, VT' = VT \cap V_i$ в новое множество правил P' входят только те правила из P, которые содержат только символы из множества $V_i, S' = S$.

3) Удаление правил с пустыми цепочками (λ-правил)

Правилами с пустыми цепочками называются правила вида $A \to \lambda$, где $A \in VN$. Обозначим W_i – специальное множество нетерминальных символов.

Шаг 1. $W_0 = \{A : (A \to \lambda) \in P\} \quad i = 1$.

Шаг 2. $W_i = W_{i-1} \cup \{A : (A \to \alpha) \in P, \alpha \in W_{i-1}^*\}$

Шаг 3. Если $W_i \neq W_{i-1}$, то *i*=*i*+1, перейти к шагу 2, иначе к шагу 4.

Шаг 4. $VN' = VN, VT' = VT$, в P' входят все правила из *P,* кроме правил вида $A \to \lambda$.

Шаг 5. Если $(A \to \lambda) \in P$ и в цепочку α входят символы из множества W_i, тогда на основе α строится множество цепочек $\{\alpha'\}$ исключением всех возможных комбинаций символов W_i; все правила вида $A \to \alpha'$ добавляются в P' (при этом надо учитывать дубликаты правил и бессмысленные правила).

Шаг 6. Если $S \in W_i$, тогда во множество VN' добавляется новый символ S', который становится целевым символом, а в P' добавляются два новых правила $S' \to \lambda | S$; иначе $S' = S$.

4) Удаление цепных правил

Цепные правила возможны только в том случае, если в КС-грамматике присутствуют правила вида $A \to B,\ A, B \in VN$.

Для их устранения для каждого нетерминального символа $X \in VN$ строится специальное множество цепных символов N^X, а затем на основании построенных множеств выполняется преобразование правил *P*.

Шаг 1. Для всех символов $X \in VN$ повторить шаги 2-4, затем перейти к шагу 5.

Шаг 2. $N_0^X = \{X\}$, $i = 1$.

Шаг 3. $N_1^X = N_{i-1}^X \cup \left\{B : (A \to B) \in P, B \in N_{i-1}^X\right\}$.

Шаг 4. Если $N_i^X \neq N_{i-1}^X$, то $i = i + 1$, перейти к шагу 3, иначе $N_i^X - \{X\}$ и продолжить цикл по шагу 1.

Шаг 5. $VN' = VN$, $VT' = VT$, в P' входят все правила из *P,* кроме правил вида $A \to B,\ S' = S$.

Шаг 6. Для всех правил $(A \to \alpha) \in P'$, если $B \in N^A$, $B \neq A$, то в P' добавляются правила вида $B \to \alpha$.

Данный алгоритм, также как и алгоритм удаления λ-правил, ведет к увеличению числа правил грамматики, но упрощает построение распознавателей.

Другая проблема Кс-грамматик заключается в содержание рекурсивных символов.

Нетерминальный символ A в КС-грамматике называется рекурсивным, если для него существует цепочка вывода вида $A \Rightarrow +\alpha A\beta$ ($\alpha, \beta \in (VT \cup VN)^*$. Если $\alpha=\lambda$ и $\beta\neq\lambda$, рекурсия называется левой, а грамматика леворекурсивной, если наоборот – праворекурсивной).

Любая КС-грамматика может быть как леворекурсивной, так и праворекурсивной, а также леворекурсивной и праворекурсивной одновременно. Существуют алгоритмы устранения рекурсии. И доказано, что их можно успешно применять для любой КС-грамматики.

Поскольку рекурсия лежит в основе правил грамматики полностью исключить ее невозможно. Можно лишь избавиться от одного вида рекурсии – левой или правой.

Существует алгоритм устранения левой рекурсии. Он работает со множеством правил P исходной грамматики **G,** ее множеством нетерминальных символов $VN = \{A_1, A_2, \ldots, A_n\}$ и двумя счетчиками i и j.

Шаг 1. i=1.

Шаг 2. Рассмотрим правила для символа A_i. Если они не содержат левой рекурсии, перенесем их в новое множество правил P' , а символ A_i добавим во множество нетерминальных символов VN'.

Иначе запишем правила для A_i в виде $A_i \rightarrow A_i\alpha_1 \mid A_i\alpha_2 \mid \ldots \mid A_i\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p$, где для любого $1 \leq j \leq p$ ни одна из цепочек β_j не начинается с символов A_k, таких, что $k \leq i$.

Вместо этого правила во множество P' запишем правила вида:

$A_i \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p \mid \beta_1 A_i' \mid \beta_2 A_i' \mid \ldots \mid \beta_p A_i'$

$A_i' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A_i' \mid \alpha_2 A_i \mid \ldots \mid \alpha_m A_i'$

Символы A_i и A_i' включаем во множество VN'.

Теперь все правила для A_i начинаются либо с терминального символа, либо с нетерминального символа A_k, такого, что $k>i$.

Шаг 3. Если i=n, то грамматика **G′** построена, перейти к шагу 6, иначе i = i+1, j=1 и перейти к шагу 4.

Шаг 4. Для символа A_j во множестве правил P′ заменить все правила вида $A_i \rightarrow A_j\alpha$, где $\alpha \in (\mathbf{VN} \cup \mathbf{VT})^*$, на правила вида $A_i \rightarrow \beta_1\alpha \mid \beta_2\ \alpha \mid \ldots \mid \beta_m\ \alpha$, причем $A_j \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m \mid$ - все правила для символа A_j.

Шаг 5. Если j = i – 1, то перейти к шагу 2, иначе j = j + 1 и перейти к шагу 4.

Шаг 6. Целевым символом грамматики **G′** становится символ A_k, соответствующий символу S исходной грамматики **G**.

Контрольные вопросы

1. Как называется символ грамматики, когда из него нельзя вывести ни одной цепочки терминальных символов?

а) бесплодным б) циклическим в) недостижимым г) цепным

2. Перечислить последовательность этапов преобразование КС-грамматики к приведенному виду.

3. Какова последовательность шагов алгоритма удаления правил с пустыми цепочками?

4. Какие цели достигаются путем преобразования КС-грамматик?

5. Каковы соотношения количества правил приведенной и неприведенной эквивалентных КС-грамматик?

6. Почему при преобразовании КС-грамматик к приведенному виду сначала необходимо удалить бесплодные символы, а потом – недостижимые?

7. С какой целью устраняется левая рекурсия в правилах грамматики?

Задание

1. Дана грамматика G({x, y, z}, {A, B, C, D, E,F, G, S}, P, S)

P:

S → xAyB | E

A → BCx | x | λ

B → ACy | y | λ

C → A | B | yA | xB | zC | xE | yE

D → z | Fy | Fx | λ

E → Ex | Ey | Ez | ED | FG | DG

F → BC | AC | DC | EC

G → Gx | Gy | Gz | GD

а) Исключить из нее цепные правила.

б) Удалить в заданной грамматике правила с цепными цепочками.

в) Удалить недостижимые символы грамматики.

г) Содержит ли грамматика бесплодные правила, если да удалить их.

д) Преобразовать грамматику к приведенному типу.

2. Дана грамматика G({“ ” , “(” , “)” , o, r, a, n, d, t, d}, {S, T, E, F}, P, S)

P:

S → S or T | T

T → T and E | E

E → not E | F

F → (S) | b

Преобразовать заданную грамматику, исключив из нее цепные правила.

3. Преобразовать грамматику, заданную в предыдущем задании, исключив из ее правил левую рекурсию.

Алгоритмы работы распознавателей

Для каждого класса КС-языков существует свой класс распознавателей, но все они функционируют на основе общих принципов, на которых основано моделирование работы МП-автоматов. Все распознаватели для КС-языков делят на две большие группы: нисходящие и восходящие. Нисходящие распознаватели просматривают входную цепочку символов слева направо и порождают левосторонний вывод. При этом дерево вывода строится распознавателем от корня к листьям. Восходящие распознаватели также просматривают входную цепочку символов слева направо, но порождают при этом правосторонний вывод. Дерево вывода строится от листьев к корню (снизу вверх).

Для моделирования работы этих двух групп распознавателей используются два алгоритма: алгоритм с подбором альтернатив – для нисходящих распознавателей, алгоритм «сдвиг-свертка» – для восходящих распознавателей. В общем случае эти два алгоритма универсальны. Они строятся на основе любой КС-грамматики после некоторых формальных преобразований и поэтому могут быть использованы для разбора цепочки любого КС-языка. В этом случае время разбора входной цепочки имеет экспоненциальную зависимость от длины цепочки. Однако для линейных распознавателей эти алгоритмы могут быть модифицированы таким образом, чтобы время разбора имело линейную зависимость от длины входной цепочки. Для каждого такого класса предусматривается своя модификация.

Восходящий синтаксический анализ, как правило, привлекательнее нисходящего. Класс языков, заданный восходящими распознавателями, значительно шире. Однако нисходящий синтаксический анализ предпочтительнее с точки зрения процесса трансляции, поскольку на его основе легче организовать процесс порождения результирующего языка.

Распознаватели с возвратом – самый примитивный тип распознавателей для КС-языков. Логика их работы основана на моделировании работы недетерминированного МП-автомата.

Нисходящий распознаватель с подбором альтернатив моделирует работу МП-автомата с одним состоянием q: R{q, V, Z, δ, S, q}. Автомат распознает цепочки КС-языка, заданного КС-грамматикой G(VT,VN, P, S).

Начальная конфигурация автомата определяется как (q, α, S) , т.е. автомат находится в своем единственном состоянии, считывающая головка находится в начале входной цепочки символов α, в стеке лежит символ, соответствующий целевому символу грамматики.

Конечная конфигурация определяется как (q, λ, λ) – автомат пребывает в своем единственном состоянии, считывающая головка располагается за концом входной цепочки, стек пуст.

Работу данного автомата можно неформально описать следующим: если на верхушке стека автомата находится нетерминальный символ A, то его можно заменить на цепочку символов α, не сдвигая считывающую головку, если в грамматике есть правило A→α. Этот шаг называется «подбор альтернативы». Если же на верхушке стека находится терминальный символ *a,* совпадающий с текущим символом входной цепочки, то этот символ можно выбросить из стека и передвинуть считывающую головку на одну позицию вправо. Этот шаг называется выброс. Данный МП-автомат может быть недетерминированным, поскольку при подборе альтернативы в грамматике может оказаться более одного правила вида A→α, и тогда функция перехода будет содержать более одного следующего состояния – несколько альтернатив МП-автомата.

Решение о том, выполнять ли на каждом шаге работы МП-автомата выброс или подбор альтернативы, принимается однозначно. Моделирующий алгоритм должен обеспечивать выбор одной из возможных альтернатив и хранение информации о том, какие альтернативы, на каком шаге уже были выбраны, чтобы иметь возможность вернуться к этому шагу и подобрать другие альтернативы.

Данный алгоритм строит левосторонние выводы и для его моделирования необходимо, чтобы грамматика не была леворекурсивной, иначе он войдет в бесконечный цикл.

Существует множество способов реализации алгоритма с подбором альтернатив, один из них рассмотрим ниже.

Для удобства работы все правила исходной грамматики G представим в виде $A\rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_1$, пронумеровав все возможные альтернативы для каждого нетерминального символа A. Входную цепочку запишем в виде $\alpha=a_1a_2\ldots a_n$. Используется дополнительное состояние b, сигнализирующее о выполнение возврата к уже прочитанной части цепочки. Обозначим через L_1 стек МП-автомата. Для хранения выбранных альтернатив используется стек L_2, который хранит символы входного языка автомата и символ A_j – показывающий, что для символа A была выбрана альтернатива с номером j. Оба стека представлены цепочкой символов. В цепочке L_1 символы помещаются слева, а в L_2 справа. Тогда состояние алгоритма на каждом шаге определяется четырьмя параметрами (Q, i, L_1, L_2), где Q - текущее состояние автомата, i – положение считывающей головки во входной цепочке.

Начальным состоянием алгоритма является состояние (q, 1, S, λ), где S - целевой символ грамматики. Алгоритм циклически выполняется следующую последовательность шагов:

Шаг 1. Разрастание. $(q, i, A\beta, \alpha) \rightarrow (q, i, \gamma_1\beta, \alpha A_1)$, если $A\rightarrow\gamma_1$ – это первая из всех возможных альтернатив для символа A.

Шаг 2. Успешное сравнение. $(q, i, a\beta, \alpha) \rightarrow (q, i + 1, \beta, \alpha a)$, если $a = a_i$, $a\in VT$.

Шаг 3. Завершение. Если состояние соответствует (q, n+1, λ, α), то разбор завершен, алгоритм завершает работу, иначе $(q, i, \lambda, \alpha) \rightarrow (b, i, \lambda, \alpha)$, когда $i \neq n+1$.

Шаг 4. Неуспешное сравнение. $(q, i, a\beta, \alpha) \rightarrow (b, i, a\beta, \alpha)$, если $a \neq a_i$, $a\in VT$.

Шаг 5. Возврат на ходу. $(b, i, \beta, \alpha a) \rightarrow (q, i - 1, a\beta, \alpha)$, для любого $a\in VT$.

Шаг 6. Другая альтернатива. Исходное состояние $(b, i, \gamma_j\beta, \alpha A_j)$, выполнить действия:

перейти в состояние $(q, i, \gamma_{j+1}\beta, \alpha A_{j+1})$, если существует альтернатива $A \rightarrow\gamma_{j+1}$ для символа $a\in VN$;

сигнализировать об ошибке и прекратить выполнение алгоритма, если $A\equiv S$ и не существует более альтернатив для символа S,

иначе перейти в состояние (q, i, Aβ, α).

В случае успешного завершения алгоритма цепочку вывода можно построить на основе содержимого стека L_2: поместить в цепочку номер правила m, соответствующего альтернативе $A \rightarrow \gamma_j$, если в стеке содержится символ A_j. Все терминальные символы стека игнорируются.

Недостаток алгоритма нисходящего разбора с возвратом – большая затрата времени, преимуществами реализации является простота и универсальность.

Данный алгоритм не применяется в реальных компиляторах, но его принципы лежат в основе многих нисходящих распознавателей, строящих левосторонние выводы и работающих без возвратов.

Восходящий распознаватель на основе алгоритма «сдвиг-свертка» строится на основе МП-автомата с одним состоянием q: R{q, V, Z, δ, S, q}. Автомат распознает цепочки КС-языка, заданного КС-грамматикой G(VT,VN, P, S).

Начальная конфигурация автомата определяется как (q, α, λ) – автомат находится в своем единственном состоянии, считывающая головка в начале входной цепочки символов α, стек пуст.

Конечная конфигурация определяется как (q, λ, S) – автомат пребывает в своем единственном состоянии, считывающая головка за концом входной цепочки, соответствующий целевому символу грамматики.

Работу данного автомата можно описать следующим: если на верхушке стека автомата находится цепочка символов γ, то ее можно заменить на нетерминальный символ A, не сдвигая считывающую головку, если в грамматике есть правило A→γ. Этот шаг называется «свертка». С другой стороны, если считывающая головка автомата обозревает некоторый символ входной цепочки *a,* то этот символ можно поместить в стек и передвинуть считывающую головку на одну позицию вправо. Этот шаг называется «сдвиг» или «перенос».

Данный алгоритм строит правосторонние выводы и для его моделирования необходимо, чтобы грамматика не содержала λ-правил и цепных правил.

Такой расширенный МП-автомат потенциально имеет больше неоднозначностей, чем алгоритм с подбором альтернатив. На каждом его шаге решаются вопросы:

какой из шагов необходимо выполнить: сдвиг или свертку;

при выполнении свертки – какую цепочку γ выбрать для поиска правил (она должна встречаться в правой части правил);

какое правило выбрать для свертки, если их окажется несколько.

Алгоритм моделирования работы расширенного МП-автомата, аналогично алгоритму нисходящего распознавателя использует дополнительное состояние b и стек возвратов L_2. В стек помещаются номера правил грамматики, используемых для свертки, или 0, если на данном шаге был выполнен сдвиг.

Состояние алгоритма на каждом шаге также определяется четырьмя параметрами (Q, i, L_1, L_2), где Q - текущее состояние автомата (q или b), i – положение считывающей головки во входной цепочке.

Начальным состоянием алгоритма является состояние (q, 1, λ, λ). Алгоритм циклически выполняется следующую последовательность шагов:

Шаг 1. Попытка свертки. (q, i, αβ, γ) → (q, i, αA, jγ), если A→ β – это первое из всех возможных правил с номером j для подцепочки β, причем оно есть первое подходящее правило для цепочки αβ, для которой правило вида A → β существует. Если удалось выполнить свертку – возвращаемся к шагу 1, иначе – выполняется шаг 2.

Шаг 2. Перенос – сдвиг. Если i< n+1, то (q, i, α, γ) → (q, i + 1, αa_i, 0 γ), $a_i \in$ VT. Если i = n+1, то перейти к шагу 3, иначе – к шагу 4.

Шаг 3. Завершение. Если состояние соответствует (q, n+1, S, γ), то разбор завершен, алгоритм завершает работу, иначе перейти к шагу 4.

Шаг 4. Переход к возврату. (q, n+1, α, γ) → (b, n+1, α, γ).

Шаг 5. Возврат. Если исходное состояние (b, i, αa, j γ), то:

перейти в состояние (q, i, α′B, kγ), если j>0, и A → β – это правило с номером j, и существует правило B → β′ с номером k, k >j, такое, что αβ = α′β′; после чего возврат к шагу 1;

перейти в состояние (b, n+1, αβ, γ), если i=n+1, j>0, A →β – это правило с номером j, и не существует других правил из множества P с номером k>j, таких, что их правая часть является правой подцепочкой из цепочки αβ; вернуться к шагу 5;

перейти в состояние (q, i+1, αβa_i, 0γ), $a_i \in$ VT, если $i \neq n+1$, j>0, A →β это правило с номером j, и не существует других правил из множества P с номером k>j, таких, что их правая часть является правой подцепочкой из цепочки αβ; перейти к шагу 1;

иначе сигнализировать об ошибке и прекратить алгоритм.

Если исходное состояние (b, i, αa, 0γ), a∈ VT, то если i>1, перейти в следующее состояние (b, i – 1, α, γ), и вернуться к шагу 5; иначе сигнализировать об ошибке и прекратить выполнение алгоритма.

В случае успешного завершения алгоритма цепочку вывода можно построить на основании содержимого стека L_2, полученного в результате выполнения алгоритма, удалив из него все цифры 0.

Стремление улучшить алгоритм с подбором альтернатив заключается в первую очередь в определении метода, по которому на каждом шаге алгоритма можно было бы однозначно выбрать одну из множества альтернатив. Наиболее очевидным здесь является ее выбор на основании терминального символа a, обозреваемого считывающей головкой на каждом шаге его работы.

Алгоритм разбора по методу рекурсивного спуска. В его реализации для каждого нетерминального символа A грамматики строится процедура разбора, которая получает на вход цепочку символов α и положение считывающей головки i в цепочке. Если для символа A в грамматике определено более одного правила A $\rightarrow a\gamma$, $a \in$ VT , $\gamma \in$ $(VT \cup VN)^*$, первый символ правой части которого совпадал бы с текущим символом входной цепочки $a=\alpha_i$. Если такого правила не найдено, то цепочка принимается. Иначе (если найдено правило A $\rightarrow a\gamma$ или для символа A в грамматике существует только одно правило A $\rightarrow \gamma$),запоминается номер правила и, когда $a = \alpha_i$, считывающая головка передвигается (увеличивается i),а для каждого нетерминального символа в цепочке γ рекурсивно вызывается процедура разбора этого символа.

Условия применимости метода можно получить из описания алгоритма – в грамматике входного языка для любого нетерминального символа A возможны только два варианта правил:

A $\rightarrow \gamma$, и это единственное правило для A;

A $\rightarrow a_1 \beta 1 \mid a_2 \beta_2 \mid \ldots \mid a_j\beta_j$, для любого i и если $i \neq j$, то $a_i \neq a_j$.

Этим условиям удовлетворяет незначительное количество реальных грамматик. Это достаточные, но не необходимые условия. Если грамматика не удовлетворяет этим условиям, еще не значит, что заданный ею язык не может распознаваться с помощью метода рекурсивного спуска. Возможно, над грамматикой необходимо выполнить ряд преобразований.

Существуют преобразования, которые способствуют приведению грамматики к требуемому виду (но не гарантируют его достижения).

Логическим продолжением идеи, лежащей в основе алгоритма рекурсивного спуска, является предложение использовать для выбора одной из многих альтернатив не один, а несколько символов входной цепочки. Существует класс грамматик, основанный на этом принципе. Это LL(k)-грамматики.

Грамматика обладает свойством LL(k), k>0, если на каждом шаге вывода для однозначного выбора очередной альтернативы достаточно знать символ на верхушке стека и рассмотреть первые k символов от текущего положения считывающей головки во входной цепочке символов.

Грамматика называется LL(k)-грамматикой, если она обладает свойством LL(k).

Название LL(k) несет определенный смысл. Первая буква – left означает, что входная цепочка символов читается в направлении слева направо. Вторая буква – left означает, что при работе распознавателя используется левосторонний вывод, а вместо буквы k в названии класса стоит число, показывающее, сколько символов надо рассмотреть.

Для LL(k)-грамматик известны следующие полезные свойства:

всякая LL(k)-грамматика для любого k>0 является однозначной;

существует алгоритм, позволяющий проверить, является ли заданная грамматика LL(k)-грамматикой для строго определенного k>0,

все грамматики, допускающие разбор по методу рекурсивного спуска являются подклассом LL(1)-грамматик, но не наоборот.

Существуют и неразрешимые проблемы LL(k)-грамматик:

не существует алгоритма, который мог бы проверить, является ли заданная КС-грамматика LL(k)-грамматикой для некоторого произвольного k;

не существует алгоритма, который мог бы преобразовать произвольную КС-грамматику к виду LL(k)-грамматики для некоторого k.

Класс LL-грамматик широк, но все же недостаточен для того, чтобы покрыть все возможные синтаксические конструкции языков программирования. Однако эти грамматики удобны для использования, поскольку позволяют строить линейные распознаватели.

Для построения распознавателей LL(k)-грамматик используются два множества:

FIRST(k, α) – множество терминальных цепочек, выводимых из $\alpha \in (VT \cup VN)^*$, укороченных до k символов,

FOLLOW(k, A) – множество укороченных до k символов терминальных цепочек, которые могут следовать непосредственно за A$\in VN$ в цепочках вывода.

Для LL(1)-грамматики алгоритм работы распознавателя заключается в двух условиях, проверяемых на шаге выбора альтернативы. На шаге выброса алгоритм остается прежним.

Исходными данными для проверки условий на шаге альтернативы являются терминальный символ, обозреваемый считывающей головкой, и нетерминальный символ A, находящийся на верхушке стека.

Необходимо выбрать в качестве альтернативы правило $A \to \alpha$, если *a* принадлежит FIRST(1, α).

Необходимо выбрать в качестве альтернативы правило $A \to \lambda$, если *a* принадлежит FOLLOW(1, A).

Если ни одно из этих условий не выполняется, то цепочка не принадлежит заданному языку, и алгоритм сообщает об ошибке.

Множество FIRST(1,α) строится сразу для всех нетерминальных символов грамматики. Предварительно из правил исходной грамматики G исключаются λ-правила, и если во вновь полученной грамматике G’ начальный символ S’ новый, при построении множества FIRST(1,α) он не учитывается.

Шаг 1. Для всех A$\in VN$:

FIRST0(1, A)= $\{X \mid A \to X\alpha \in P, X \in (VT \cup VN, \alpha \in (VT \cup VN)^*\}$

(т.е. первоначально вносим во множество первых символов для каждого нетерминального символа A все символы, стоящие в начале правых частей правил для этого символа A). i=0.

Шаг 2. Для всех нетерминальных символов A: FIRST1(1,A)= FIRSTi(1,A)$\cup$ FIRSTi(1,B), B$\in$ (FIRSTi(1,A) $\cap$ VN).

Шаг 3. Если существует нетерминальный символ A такой, что FIRSTi+1(1,A) ≠ FIRSTi(1,A), то i = i+1 и вернуться к шагу 2, иначе перейти к шагу 4.

Шаг 4. Для всех нетерминальных символов A FIRST(1,A)= FIRSTi(1,A)\VN (исключаем из построенных множеств все нетерминальные символы).

Множество FOLLOW(1,A) строится сразу для всех нетерминальных символов грамматики. Предварительно строятся все множества FIRSTi(1,A).

Шаг 1. Первоначально во множество последующих символов FOLLOW0(1, A) для каждого нетерминального символа A все символы, которые в правых частях правил встречаются непосредственно за A; i=0.

Шаг 2. FOLLOW0(1,S)=FOLLOW0(1,S)$\cup$ λ (вносим пустую цепочку во множество последующих символов для целевого символа – это означает, что в конце разбора за целевым символом цепочка кончается, иногда для этого используют и специальный символ конца цепочки).

Шаг 3. Для всех нетерминальных символов A

FOLLOWi′(1,A)=FOLLOWi(1, A)$\cup$ FIRSTi(1,B) для всех нетерминальных символов B$\in$ (FOLLOWi(1,A) $\cap$ VN).

Шаг 4. Для всех нетерминальных символов A

FOLLOWi″(1,A)=FOLLOWi′(1,A)$\cup$ FOLLOWi′(1,B),

B$\in$ (FOLLOWi′(1,A) $\cap$ VN), если существует правило B$\rightarrow\lambda$.

Шаг 5. Для всех нетерминальных символов A

FOLLOWi+1(1,A)=FOLLOWi″(1,A)$\cup$ FOLLOWi″(1,B), если существует правило B$\rightarrow\alpha$A, для $\alpha \in (VT \cup VN)^*$.

Шаг 6. Если существует такой нетерминальный символ A, что FOLLOWi+1(1,A) $\neq$ FOLLOWi(1,A), то i=i+1 вернуться к шагу 3, иначе перейти к шагу 7.

Шаг 7. Для всех нетерминальных символов A FOLLOWi(1,A) = FOLLOWi(1,A)\VN (исключаем из построенных множеств все нетерминальные символы).

Рассмотренный алгоритм эффективен, но жесткие ограничения на правила LL(1)-грамматик сужают его возможности.

Контрольные вопросы

1. В чем отличие нисходящего распознавателя и восходящего?

2. Существуют ли ограничения, накладываемые на КС-грамматику, при построении нисходящего распознавателя с подбором альтернатив для ее цепочек? Если да, в чем их суть?

3. Какими параметрами определяется состояние алгоритма нисходящего распознавателя с подбором альтернатив на каждом шаге?

4. Используется ли алгоритм нисходящего распознавателя с подбором альтернатив в реальных компиляторах.

5. Каково начальное состояние алгоритма «сдвиг-свертка»?

6. Каковы условия применимости метода рекурсивного спуска?

7. За счет чего класс LL(1)-грамматик является более широким, чем класс КС-грамматик, для которых можно построить распознаватель по методу рекурсивного спуска?

8. В чем заключаются ограничения на правила LL(1)-грамматик?

Задание

1. Построить нисходящий распознаватель с возвратом для грамматики, заданной в задании 2 предыдущего раздела.

2. Построить распознаватель, основанный на методе рекурсивного спуска для грамматики G({a, b, c}, {S, A, B, C}, P, S)

P:

S→ aABb | bBAa | cCc

A→ aA | bB | cC

B → b | aAC

C→ aA | bB| cC

3. На основании распознавателя, построенного в предыдущем задании выполнить разбор цепочки символов aabbabbcabbb.

СОДЕРЖАНИЕ

Библиографический список

1. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. М.: Издательский дом «Вильямс», 2003. – 768 с.

2. Галаган Т.А., Соловцова Л.А. Практикум по лингвистическим основам информатики. Благовещенск: Изд-во АмГУ, 2005. – 98 с.

3. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. СПб.: Питер, 2003. – 736 с.

4. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. М.: МИР, 1979. – 650 с.

5. Молчанов А.Ю. Системное программное обеспечение. СПб.: Питер, 2006. – 396 с.

Татьяна Алексеевна Галаган,
доцент кафедры ИиУС АмГУ

Изд-во АмГУ. Подписано к печати Формат 60x84/16. Усл. печ. л.
Тираж Заказ