Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный
университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

ТРАНСЛЯЦИЯ ФОРМАЛЬНЫХ ЯЗЫКОВ

Курс лекций

Витебск
ВГУ имени П.М. Машерова
2023

УДК 004.4’412(075.8)
ББК 32.973.02я73
Т65

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 4 от 27.12.2022.

Составитель: старший преподаватель кафедры прикладного и системного программирования ВГУ имени П.М. Машерова, магистр физико-математических наук **С.В. Сергеенко**

Р е ц е н з е н т ы :
заведующий кафедрой инженерной физики ВГУ имени П.М. Машерова, кандидат физико-математических наук *А.И. Никитин;*
заведующий кафедрой математики и информационных технологий УО «ВГТУ», кандидат физико-математических наук, доцент *Т.В. Никонова*

Трансляция формальных языков : курс лекций / сост. С.В. Сергеенко. – Витебск : ВГУ имени П.М. Машерова, 2023. – 90 с.
Т65 ISBN 978-985-30-0052-8.

В данном учебном издании излагаются вопросы трансляции формальных языков: фазы трансляции, построение ДКА, синтаксический и контекстный анализ, генерация промежуточного представления и целевого кода, оптимизация.

Предназначается для студентов специальностей «Прикладная информатика (веб-программирование и компьютерный дизайн)» (дисциплина «Теория автоматов и формальных языков»), «Программное обеспечение информационных технологий» (дисциплина «Низкоуровневое программирование и трансляторы»).

УДК 004.4’412(075.8)
ББК 32.973.02я73

ISBN 978-985-30-0052-8

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В курсе лекций излагаются базовые вопросы трансляции формальных языков, включая основы теории автоматов и формальных языков, фазы трансляции (в том числе для языков программирования), построения детерминированных конечных автоматов с помощью регулярных выражений, нисходящего и восходящего синтаксического анализа, синтеза и выведения типов, проверки правильности употребления имен и передачи управления, генерации промежуточного представления и целевого кода оптимизации.

Кроме общих вопросов трансляции формальных языков рассматриваются особенности, связанные с языками программирования.

Предназначается для студентов специальностей «Прикладная информатика (веб-программирование и компьютерный дизайн)» (дисциплина «Теория автоматов и формальных языков»), «Программное обеспечение информационных технологий» (дисциплина «Низкоуровневое программирование и трансляторы»).

1. ОСНОВНЫЕ ПОНЯТИЯ ТЕОРИИ АВТОМАТОВ И ФОРМАЛЬНЫХ ЯЗЫКОВ

1.1. Алфавит, цепочки символов и операции над ними

Алфавит, или словарь – это конечное множество символов. Для обозначения символов мы будем пользоваться цифрами, латинскими буквами и специальными литерами типа #, $. Символом называется любой элемент алфавита.

Пусть V – алфавит. Цепочка в алфавите V – это любая строка конечной длины, составленная из символов алфавита V. Синонимом цепочки являются предложение, строка и слово. Пустая цепочка (обозначается ε) – это цепочка, в которую не входит ни один символ.

Конкатенацией цепочек x и y называется цепочка xy. Заметим, что xε = εx = x для любой цепочки x.

Пусть x, y, z – произвольные цепочки в некотором алфавите. Цепочка y называется подцепочкой цепочки xyz. Цепочки x и y называются, соответственно, префиксом и суффиксом цепочки xy. Заметим, что любой префикс или суффикс цепочки является подцепочкой этой цепочки. Кроме того, пустая цепочка является префиксом, суффиксом и подцепочкой для любой цепочки.

Введем обозначение x^n = x … x – конкатенация n цепочек x.

Длиной цепочки w (обозначается |w|) называется число символов в ней. Например, |abababa| = 7, а |ε| = 0.

1.2. Формальный язык. Операции над формальными языками

Язык в алфавите V – это некоторое множество цепочек в алфавите V.

Пример Пусть дан алфавит V = {a, b}. Вот некоторые языки в алфавите V:

- $L_1 = \emptyset$ – пустой язык;
- $L_2 = \{\varepsilon\}$ – язык, содержащий только пустую цепочку (заметим, что L_1 и L_2 – различные языки);
- $L_3 = \{\varepsilon, a, b, aa, ab, ba, bb\}$ – язык, содержащий цепочки из a и b, длина которых не превосходит 2;
- L_4 – язык, включающий всевозможные цепочки из a и b, содержащие четное число a и четное число b;
- $L_5 = \{a^{nn} \mid n > 0\}$ – язык цепочек из a, длины которых представляют собой квадраты натуральных чисел.

Два последних языка содержат бесконечное число цепочек.

Введем обозначение V^* для множества всех цепочек в алфавите V, включая пустую цепочку. Каждый язык в алфавите V является

подмножеством V^*. Для обозначения множества всех цепочек в алфавите V, кроме пустой цепочки, будем использовать V^+.

Пример Пусть V = {0, 1}. Тогда V^* = {ε, 0, 1, 00, 01, 10, 11, 000, …}, V^+ = {0, 1, 00, 01, 10, 11, 000, …}.

Введем некоторые операции над языками.

Так как язык является множеством, то все операции, определенные на множествах, также будут определены и для языков: объединение, пересечение, разность, симметрическая разность. Формально, можно говорить и о декартовом произведении языков, но на практике такая операция обычно используется редко.

Пусть L_1 и L_2 – языки в алфавите V. Конкатенацией языков L_1 и L_2 называется язык L_1 L_2 = {xy|x∈L_1, y∈L_2}.

Пример Пусть L_1 = {aa, bb} и L_2 = {ε, a, bb}. Тогда L_1L_2 = {aa, bb, aaa, bba, aabb, bbbb}.

Пример Пусть L_1 = {a, ab} и L_2 = {c, bc}. Тогда L_1L_2 = {ac, abc, abbc}.

Пусть L – язык в алфавите V. Введём обозначения: L^0 = {ε}; $L^n = LL^{n-1}$, если n > 1. Итерацией (замыканием Клини) языка L называется язык L^*, определяемый следующим образом:

$$L^* = L^0 \cup L^1 \cup \ldots \cup L^n \cup \ldots = \bigcup_{n=0}^{\infty} A_n.$$

Пример Пусть L = {aa, bb}. Тогда L^* = {ε, aa, bb, aaaa, aabb, bbaa, bbbb, aaaaaa, ...}.

Большинство языков, представляющих интерес, содержат бесконечное число цепочек. При этом возникают три важных вопроса.

Во-первых, как представить язык (то есть специфицировать входящие в него цепочки)? Если язык содержит только конечное множество цепочек, ответ прост. Можно просто перечислить его цепочки. Если язык бесконечен, необходимо найти для него конечное представление. Это конечное представление, в свою очередь, будет строкой символов над некоторым алфавитом вместе с некоторой интерпретацией, связывающей это представление с языком.

Во-вторых, для любого ли языка существует конечное представление? Можно предположить, что ответ отрицателен. Мы увидим, что множество всех цепочек над алфавитом счетно. Язык – это любое подмножество цепочек. Из теории множеств известно, что множество всех подмножеств счетного множества несчетно. Хотя мы и не дали строгого определения того, что является конечным представлением, интуитивно ясно, что любое разумное определение конечного представления ведет только к счетному множеству конечных представлений, поскольку нужно иметь возможность записать такое конечное представление в виде строки символов конечной длины. Поэтому языков значительно больше, чем конечных представлений.

В-третьих, можно спросить, какова структура тех классов языков, для которых существует конечное представление?

1.3. Представление языков. Способы описания языка. Их эквивалентность

Процедура – это конечная последовательность инструкций, которые могут быть механически выполнены. Примером может служить машинная программа. Процедура, которая всегда заканчивается, называется алгоритмом.

Один из способов представления языка – дать алгоритм, определяющий, принадлежит ли цепочка языку. Более общий способ состоит в том, чтобы дать процедуру, которая останавливается с ответом “да” для цепочек, принадлежащих языку, и либо останавливается с ответом “нет”, либо вообще не останавливается для цепочек, не принадлежащих языку. Говорят, что такая процедура или алгоритм распознает язык.

Такой метод представляет язык с точки зрения распознавания. Язык можно также представить методом порождения. А именно, можно дать процедуру, которая систематически порождает в определенном порядке цепочки языка.

Если мы можем распознать цепочки языка над алфавитом V либо с помощью процедуры, либо с помощью алгоритма, то мы можем и генерировать язык, поскольку мы можем систематически генерировать все цепочки из $V*$, проверять каждую цепочку на принадлежность языку и выдавать список только цепочек языка. Но если процедура не всегда заканчивается при проверке цепочки, мы не сдвинемся дальше первой цепочки, на которой процедура не заканчивается. Эту проблему можно обойти, организовав проверку таким образом, чтобы процедура никогда не продолжала проверять одну цепочку бесконечно. Для этого введем следующую конструкцию.

Предположим, что V имеет p символов. Мы можем рассматривать цепочки из V^* как числа, представленные в базисе p, плюс пустая цепочка ε. Можно занумеровать цепочки в порядке возрастания длины и в “числовом” порядке для цепочек одинаковой длины.

Пусть P – процедура для проверки принадлежности цепочки языку L. Предположим, что P может быть представлена дискретными шагами, так что имеет смысл говорить об i-ом шаге процедуры для любой данной цепочки. Прежде чем дать процедуру перечисления цепочек языка L, дадим процедуру нумерации пар положительных чисел.

Все упорядоченные пары положительных чисел (x, y) можно отобразить на множество положительных чисел следующей формулой:

$$z = (x + y - 1)(x + y - 2)/2 + y$$

Пары целых положительных чисел можно упорядочить в соответствии со значением z.

Теперь можно дать процедуру перечисления цепочек L. Нумеруем упорядоченные пары целых положительных чисел – (1,1), (2,1), (1,2), (3,1), (2,2), ... При нумерации пары (i, j) генерируем i-ю цепочку из V^* и применяем к цепочке первые j шагов процедуры P. Как только мы определили, что

сгенерированная цепочка принадлежит L, добавляем цепочку к списку элементов L. Если цепочка i принадлежит L, это будет определено P за j шагов для некоторого конечного j. При перечислении (i, j) будет сгенерирована цепочку с номером i. Легко видеть, что эта процедура перечисляет все цепочки L.

Если мы имеем процедуру генерации цепочек языка, то мы всегда можем построить процедуру распознавания предложений языка, но не всегда алгоритм. Для определения того, принадлежит ли x языку L, просто нумеруем предложения L и сравниваем x с каждым предложением. Если сгенерировано x, процедура останавливается, распознав, что x принадлежит L. Конечно, если x не принадлежит L, процедура никогда не закончится. Однако, если удастся расширить порядок порождения L до порядка порождения V^*, то процедуру можно останавливать, определив, что x не принадлежит L, в тот момент, когда очередная сгенерированная цепочка языка L будет следовать после x в порядке полученном порядке порождения V^*.

Язык, предложения которого могут быть сгенерированы процедурой, называется рекурсивно перечислимым. Язык рекурсивно перечислим, если имеется процедура, распознающая предложения языка. Говорят, что язык рекурсивен, если существует алгоритм для распознавания языка. Класс рекурсивных языков является собственным подмножеством класса рекурсивно перечислимых языков. Мало того, существуют языки, не являющиеся даже рекурсивно перечислимыми.

В этом курсе лекций рассматриваются различные способы описания (представления) формальных языков, принадлежащих различным классам: регулярные выражения, порождающие грамматики, разбирающие выражения грамматики, конечные автоматы, автоматы с магазинной памятью. Два способа описания языка будем называть эквивалентными, если они являются представлениями одного и того же языка.

1.4. Порождающие грамматики. Классификация порождающих грамматик и формальных языков

Для нас наибольший интерес представляет одна из систем генерации языков – грамматики. Понятие грамматики изначально было формализовано лингвистами при изучении естественных языков. Предполагалось, что это может помочь при их автоматической трансляции. Однако, наилучшие результаты в этом направлении достигнуты при описании не естественных языков, а языков программирования. Примером может служить способ описания синтаксиса языков программирования при помощи БНФ – формы Бэкуса–Наура.

Грамматика – это четверка $G = (N, T, P, S)$, где N – алфавит нетерминальных символов; T – алфавит терминальных символов, $N \cap T = \emptyset$; P – конечное множество правил вида $\alpha \rightarrow \beta$, где $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$; $S \in N$ – начальный символ (или аксиома) грамматики.

Мы будем использовать большие латинские буквы для обозначения нетерминальных символов, малые латинские буквы из начала алфавита для обозначения терминальных символов, малые латинские буквы из конца алфавита для обозначения цепочек из T^* и, наконец, малые греческие буквы для обозначения цепочек из $(N \cup T)^*$.

Будем использовать также сокращенную запись $A \to \alpha_1|\alpha_2 \mid \ldots \mid\alpha_n$ для обозначения группы правил $A \to \alpha_1$, $A \to \alpha_2$, … , $A \to \alpha_n$.

Определим на множестве $(N \cup T)^*$ бинарное отношение выводимости $\Rightarrow$ следующим образом: если $\delta \to \gamma \in P$, то $\alpha\delta\beta \Rightarrow \alpha\gamma\beta$ для всех $\alpha, \beta \in (N \cup T)^*$. Если $\alpha_1 \Rightarrow \alpha_2$, то говорят, что цепочка α_2 непосредственно выводима из α_1.

Мы будем использовать также рефлексивно-транзитивное и транзитивное замыкания отношения $\Rightarrow$, а также его степень $k > 0$ (обозначаемые соответственно $\Rightarrow^*$, $\Rightarrow^+$ и $\Rightarrow^k$). Если $\alpha_1 \Rightarrow^* \alpha_2$ ($\alpha_1 \Rightarrow^+ \alpha_2$, $\alpha_1 \Rightarrow^k \alpha_2$), то говорят, что цепочка α_2 выводима (нетривиально выводима, выводима за k шагов) из α_1.

Если $\alpha \Rightarrow^k \beta$ $(k > 0)$, то существует последовательность шагов

$$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{k-1} \Rightarrow \gamma_k$$

где $\alpha = \gamma_0$ и $\beta = \gamma_k$. Последовательность цепочек γ_0 , γ_1, γ_2 , ... , γ_k в этом случае называют выводом β из α.

Сентенциальной формой грамматики G называется цепочка, выводимая из ее начального символа.

Языком, порождаемым грамматикой G (обозначается L(G)), называется множество всех ее терминальных сентенциальных форм, то есть

$$L(G) = \{w|w \in T^*, S \Rightarrow^+ w\}$$

Грамматики G_1 и G_2 называются эквивалентными, если они порождают один и тот же язык, то есть $L(G_1) = L(G_2)$.

Пример Грамматика $G = (\{S, B, C\}, \{a, b, c\}, P, S)$, где $P = \{S \to aSBC, S \to aBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc\}$, порождает язык $L(G) = \{a^n b^n c^n \mid n > 0\}$.

Действительно, применяем $n - 1$ раз правило 1 и получаем $a^{n-1}S(BC)^{n-1}$, затем один раз правило 2 и получаем $a^n(BC)^n$, затем $n(n - 1)/2$ раз правило 3 и получаем $a^nB^nC^n$.

Затем используем правило 4 и получаем $a^nbB^{n-1}C^n$. Затем применяем $n-1$ раз правило 5 и получаем $a^nb^nC^n$. Затем применяем правило 6 и $n-1$ раз правило 7 и получаем $a^nb^nc^n$. Можно показать, что язык L(G) состоит из цепочек только такого вида.

Пример Рассмотрим грамматику $G = (\{S\}, \{0, 1\}, \{S \to 0S1, S \to 01\}, S)$. Легко видеть, что цепочка $000111 \in L(G)$, так как существует вывод

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111$$

Нетрудно показать, что грамматика порождает язык $L(G) = \{0^n1^n \mid n > 0\}$.

Пример Рассмотрим грамматику $G = (\{S, A\}, \{0, 1\}, \{S \to 0S, S \to 0A, A \to 1A, A \to 1\}, S)$. Нетрудно показать, что грамматика порождает язык $L(G) = \{0^n1^m \mid n, m > 0\}$.

Иерархия Хомского – классификация формальных языков и формальных грамматик, согласно которой они делятся на 4 типа по их условной сложности. Предложена профессором Массачусетского технологического института, лингвистом Ноамом Хомским.

Согласно Хомскому, формальные грамматики можно разделить на четыре типа. Для отнесения грамматики к тому или иному типу необходимо соответствие всех её правил (продукций) некоторым схемам.

Тип 0 – неограниченные. Грамматика с фразовой структурой G – это алгебраическая структура, упорядоченная четвёрка (V_T, V_N, P, S), где: V_T – алфавит (множество) терминальных символов – терминалов, V_N – алфавит (множество) нетерминальных символов – нетерминалов, $V = V_T \cup V_N$ – словарь G, причём $V_T \cap V_N = \emptyset$; P – конечное множество продукций (правил) грамматики, $P \subseteq V^+ \times V^*$; S – начальный символ (источник).

К типу 0 по классификации Хомского относятся неограниченные грамматики – грамматики с фразовой структурой, то есть все без исключения формальные грамматики. Правила можно записать в виде:

$$\alpha \rightarrow \beta,$$

где $\alpha \in V^+$ – любая непустая цепочка, содержащая хотя бы один нетерминальный символ, а $\beta \in V^*$– любая цепочка символов из алфавита.

Практического применения в силу своей сложности такие грамматики не имеют.

Тип 1 – контекстно-зависимые. К этому типу относятся контекстно-зависимые (КЗ) грамматики и неукорачивающие грамматики. Для грамматики $G=(V_T, V_N, P, S)$, $V = V_T \cup V_N$ все правила имеют вид:

$\alpha A \beta \rightarrow \alpha \gamma \beta$, где $\alpha, \beta \in V^*$, $\gamma \in V^+$, $A \in V_N$. Такие грамматики относят к контекстно-зависимым.

$\alpha \rightarrow \beta$, где $\alpha, \beta \in V^+$, $1 \leq |\alpha| \leq |\beta|$. Такие грамматики относят к неукорачивающим.

Эти классы грамматик эквивалентны. Могут использоваться при анализе текстов на естественных языках, однако при построении компиляторов практически не используются в силу своей сложности. Для контекстно-зависимых грамматик доказано утверждение: по некоторому алгоритму за конечное число шагов можно установить, принадлежит цепочка терминальных символов данному языку или нет.

Тип 2 – контекстно-свободные. К этому типу относятся контекстно-свободные (КС) грамматики. Для грамматики $G = (V_T, V_N, P, S)$, $V = V_T \cup V_N$ все правила имеют вид:

$A \rightarrow \beta$, где $\beta \in V^+$ (для неукорачивающих КС-грамматик) или $\beta \in V^*$ (для укорачивающих), $A \in V_N$. То есть грамматика допускает появление в левой части правила только нетерминального символа.

КС-грамматики широко применяются для описания синтаксиса компьютерных языков.

Тип 3 – регулярные. К третьему типу относятся регулярные грамматики (автоматные) – самые простые из формальных грамматик. Они являются контекстно-свободными, но с ограниченными возможностями.

Все регулярные грамматики могут быть разделены на два эквивалентных класса, которые для грамматики вида III будут иметь правила следующего вида:

$A \to B\,\gamma$ или $A \to \gamma$, где $\gamma \in V_T^*$, $A, B \in V_N$ (для леволинейных грамматик).

$A \to \gamma\,B$ или $A \to \gamma$, где $\gamma \in V_T^*$, $A, B \in V_N$ (для праволинейных грамматик).

Регулярные грамматики применяются для описания простейших конструкций: идентификаторов, строк, констант, а также языков ассемблера, командных процессоров и др.

Формальные языки классифицируются в соответствии с типами грамматик, которыми они задаются. Однако, один и тот же язык может быть задан разными грамматиками, относящимися к разным типам. В таком случае, считается, что язык относится к наиболее простому из них. Так, язык, описанный грамматикой с фразовой структурой, контекстно-зависимой и контекстно-свободной грамматиками, будет контекстно-свободным.

Так же, как и для грамматик, сложность языка определяется его типом. Наиболее сложные – языки с фразовой структурой (сюда можно отнести естественные языки), далее – КЗ-языки, КС-языки и самые простые – регулярные языки.

Язык, порождаемый грамматикой типа i, называют языком типа i. Язык типа 0 называют также языком без ограничений, язык типа 1 – контекстно-зависимым (КЗ), язык типа 2 – контекстно-свободным (КС), язык типа 3 – праволинейным.

Каждый контекстно-свободный язык может быть порожден неукорачивающей грамматикой.

Пусть K_i – класс всех языков типа i. Доказано, что справедливо следующее (строгое) включение: $K_3 \subset K_2 \subset K_1 \subset K_0$.

Заметим, что если язык порождается некоторой грамматикой, это не означает, что он не может быть порожден грамматикой с более сильными ограничениями на правила. Приводимый ниже пример иллюстрирует этот факт.

Пример Рассмотрим грамматику $G = (\{S, A, B\}, \{0, 1\}, \{S \to AB, A \to 0A, A \to 0, B \to 1B, B \to 1\}, S)$. Эта грамматика является контекстносвободной. Легко показать, что $L(G) = \{0^n1^m | n, m > 0\}$. Однако, в предыдущем примере приведена праволинейная грамматика, порождающая тот же язык.

Существует алгоритм, позволяющий для произвольного КЗ-языка L в алфавите T , и произвольной цепочки $w \in T^*$ определить, принадлежит ли w языку L. То есть, каждый контекстно-зависимый язык является рекурсивным языком.

1.5. Автоматы как абстрактная модель

Абстрактный автомат – математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. На вход этому устройству поступают символы одного алфавита, на выходе оно выдаёт символы (в общем случае) другого алфавита.

Формально абстрактный автомат определяется как пятёрка

$$A = (S, X, Y, \delta, \lambda),$$

где S – множество состояний автомата, X, Y – конечные входной и выходной алфавиты соответственно, из которых формируются строки, считываемые и выдаваемые автоматом, δ: $S \times X \rightarrow S$ – функция переходов, λ: $S \times X \rightarrow Y$ – функция выходов.

Абстрактный автомат с выделенным начальным состоянием называется инициальным автоматом. Таким образом, абстрактный автомат определяет семейство инициальных автоматов

$$(s_i, A),\ s_i \in S$$

Если функции переходов и выходов однозначно определены для каждой пары $(s, x) \in S \times X$, то автомат называют детерминированным. В противном случае автомат называют недетерминированным или частично определённым. Для недетерминированного автомата можно считать, что значениями функций переходов и выходов являются множества состояний и символов выходного алфавита соответственно.

Если функция переходов и/или функция выходов являются случайными, то автомат называют вероятностным.

Ограничение числа состояний абстрактного автомата определило такое понятие как конечный автомат.

Функционирование автомата состоит в порождении двух последовательностей: последовательности очередных состояний автомата $s1, s_2, s_3, \ldots$ и последовательности выходных символов $y_1, y_2, y_3, \ldots$, которые для последовательности символов $x_1, x_2, x_3, \ldots$ разворачиваются в моменты дискретного времени $t = 1, 2, 3, \ldots$ Моменты дискретного времени получили название тактов.

Функционирование автомата в дискретные моменты времени t может быть описано системой рекуррентных соотношений: $s(t+1) = \delta(s(t), x(t))$; $y(t) = \lambda(s(t), x(t))$.

Для уточнения свойств абстрактных автоматов введена классификация.

Абстрактные автоматы образуют фундаментальный класс дискретных моделей как самостоятельная модель, и как основная компонента машин Тьюринга, автоматов с магазинной памятью, конечных автоматов и других преобразователей информации.

Модель абстрактного автомата широко используется как базовая для построения дискретных моделей автоматов, распознающих, порождающих и преобразующих последовательности символов.

2. ОБЗОР ПРОЦЕССА ТРАНСЛЯЦИИ

2.1. Трансляторы, интерпретаторы, компиляторы

Программы, написанные на языках программирования высокого уровня (или проблемно-ориентированных языках), перед выполнением на ЭВМ должны транслироваться в эквивалентные программы, написанные в машинном коде. Языки высокого уровня появились в середине 50-х годов, примерами первых таких языков могут служить Фортран и Кобол, а примерами недавно созданных – Алгол 68, Паскаль и Ада. Программа, которая транслирует любую программу, написанную на конкретном языке высокого уровня, в эквивалентную программу на другом языке (обычно это код машины), называется компилятором. Компилирование программы включает анализ – определение предусмотренного результата действия программы и последующий синтез – генерирование эквивалентной программы в машинном коде. В процессе анализа компилятор должен выяснить, является ли входная программа недействительной в каком-либо смысле (т.е. принадлежит ли она к языку, для которого написан данный компилятор), и если она окажется недействительной, – выдать соответствующее сообщение программисту. Этот аспект компиляции называется обнаружением ошибок.

В построении компиляторов достигнуты значительные успехи. Первые компиляторы использовали специальные методы, были медленными и не носили структурного характера. В современных компиляторах применяются более системные методы; эти компиляторы относительно быстрые и строятся таким образом, чтобы как можно более четко выделять отдельные аспекты компиляции.

Существует и другой подход. Он состоит в том, чтобы вместо трансляции каждой программы в машинный код и последующего ее выполнения программа сначала транслировалась на промежуточный язык, а затем транслировался и выполнялся каждый оператор промежуточного языка (когда он встретится). Программа, транслирующая и выполняющая программу, написанную на таком языке высокого уровня, называется интерпретатором. Применение интерпретатора вместо компилятора имеет следующие преимущества:

Передавать сообщения об ошибках пользователю часто бывает легче в терминах оригинальной программы.

Версия программы на промежуточном языке нередко оказывается компактнее, чем машинный код, выданный компилятором.

Изменение части программы не требует перекомпиляции всей программы.

Диалоговые языки, такие, как Бейсик, часто реализуются посредством интерпретатора. Промежуточным языком обычно служит некая форма обратной польской записи. Хорошим пособием по реализации диалоговых

языков может быть работа. Основной недостаток интерпретаторов заключается в том, что эти программы обычно пропускаются относительно медленно, так как операторы промежуточного кода должны транслироваться всякий раз, когда они выполняются, хотя временные издержки не так уж велики (они зависят от конструкции промежуточного языка). Применяется метод смешанного кода, в котором наиболее часто выполняемая часть интерпретируется. При этом экономится память, поскольку часть программы, которая должна интерпретироваться, будет, по всей вероятности, намного компактнее, чем скомпилированный код. Развитием этой идеи является «отбрасывающее» компилирование.

Здесь речь идет в основном о компиляторах, а не об интерпретаторах, хотя многие рассматриваемые принципы применимы и к тем, и к другим. Говоря о компиляторах, мы будет называть компилируемую программу исходной программой (или исходным текстом), а выдаваемый машинный код – объектным кодом. Аналогичным образом язык высокого уровня может называться исходным языком, а машинный – объектным языком.

Препроцессор может также раскрывать сокращения, именуемые макросами, в инструкции исходного языка. Модифицированная исходная программа затем передается компилятору. Компилятор может выдать в качестве выходных данных программу на языке ассемблера, поскольку ассемблерный код легче создать и проще отлаживать. Язык ассемблера затем обрабатывается программой, которая называется ассемблер, и дает в качестве выходных данных перемещаемый машинный код. Большие программы зачастую компилируются по частям, так что перемещаемый машинный код должен быть скомпонован совместно с другими перемещаемыми объектными файлами и библиотечными файлами в код, который можно будет выполнять на данной машине. Компоновщик («линкер») выполняет разрешение внешних адресов памяти, по которым код из одного файла может обращаться к информации из другого файла. Загрузчик затем помещает все выполнимые объектные файлы в память для выполнения.

2.2. Стадии работы компилятора

Работа компилятора складывается из двух основных этапов. Сначала он распознает структуру и значение программы, которую должен компилировать, а затем выдает эквивалентную программу в машинном коде (или коде сборки). Эти два этапа – анализ и синтез.

По идее анализ должен проводиться перед синтезом, но на практике они могут выполняться почти параллельно. Определив исходный язык, мы тем самым задаем значение его каждой допустимой конструкции (но не недопустимой). После того как анализатор распознает все конструкции в программе, он может установить, каким должен быть результат действия этой программы. Затем синтезатор вырабатывает соответствующий объектный код.

Анализ разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Затем он использует эту структуру для создания промежуточного представления исходной программы. Если анализ обнаруживает,что исходная программа неверно составлена синтаксически либо дефектна семантически, он должен выдать информативные сообщения об этом, чтобы пользователь мог исправить обнаруженные ошибки. Анализ также собирает информацию об исходной программе и сохраняет ее в структуре данных, именуемой таблицей символов, которая передается вместе с промежуточным представлением синтезу.

Синтез строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов. Анализ часто называют начальной стадией (front end), а синтез – заключительной (back end).

Если рассмотреть процесс компиляции более детально, можно увидеть, что он представляет собой последовательность фаз, каждая из которых преобразует одно из представлений исходной программы в другое. Типичное разложение компилятора на фазы: лексический анализ, синтаксический анализ, семантический анализ, генерация промежуточного кода, машинно-независимая оптимизация кода, генерация кода, машинно-зависимая оптимизация кода. На практике некоторые фазы могут объединяться, а межфазное промежуточное представление может не строиться явно. Таблица символов, в которой хранится информация обо всей исходной программе, используется всеми фазами компилятора. Некоторые компиляторы содержат фазу машинно-независимой оптимизации между анализом и синтезом. Назначение этой оптимизации – преобразовать промежуточное представление, чтобы синтез мог получить более качественную целевую программу по сравнению с той, которая может быть получена из неоптимизированного промежуточного представления. Поскольку оптимизация необязательна, некоторые фазы оптимизации в компиляторе могут отсутствовать.

Первая фаза компиляции называется лексическим анализом или сканированием. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности,называющиеся лексемами. Для каждой лексемы анализатор строит выходной токен (token) вида

⟨имя токена, значение_атрибута⟩

Он передается последующей фазе, синтаксическому анализу. Первый компонент токена, имя_токена, представляет собой абстрактный символ,использующийся во время синтаксического анализа, а второй компонент, значение атрибута,указывает на запись в таблице символов, соответствующую данному токену. Информация из записи в таблице символов необходима для семантического анализа и генерации кода. Предположим, например, что исходная программа содержит инструкцию присваивания

position = initial + rate * 60

Символы в этом присваивании могут быть сгруппированы в следующие лексемы и отображены в следующие токены, передаваемые синтаксическому анализатору.

1. position представляет собой лексему, которая может отображаться в токен ⟨id, 1⟩, где id – абстрактный символ, обозначающий идентификатор,а 1 указывает запись в таблице символов для position. Запись таблицы символов для некоторого идентификатора хранит информацию о нем,такую как его имя и тип.
2. Символ присваивания = представляет собой лексему, которая отображается в токен ⟨=⟩. Поскольку этот токен не требует значения атрибута, второй компонент данного токена опущен. В качестве имени токена может быть использован любой абстрактный символ, например такой, как assign, но для удобства записи мы будем использовать в качестве имени абстрактного символа саму лексему.
3. initial представляет собой лексему, которая отображается в токен ⟨id, 2⟩, где 2 указывает на запись в таблице символов для initial.
4. + является лексемой, отображаемой в токен ⟨+⟩.
5. rate – лексема, отображаемая в токен ⟨id, 3⟩, где 3 указывает на запись в таблице символов для rate.
6. * – лексема, отображаемая в токен ⟨*⟩.
7. 60 – лексема, отображаемая в токен.

Пробелы, разделяющие лексемы, лексическим анализатором отбрасываются.

Вторая фаза компилятора – синтаксический анализ или разбор (parsing). Анализатор использует первые компоненты токенов, полученных при лексическом анализе, для создания древовидного промежуточного представления, которое описывает грамматическую структуру потока токенов. Типичным представлением является синтаксическое дерево, в котором каждый внутренний узел представляет операцию, а дочерние узлы – аргументы этой операции.

Последующие фазы компилятора используют грамматическую структуру,которая помогает проанализировать исходную и сгенерировать целевую программу.

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице символов для последующего использования в процессе генерации промежуточного кода.

Важной частью семантического анализа является проверка типов, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа. Например, многие определения языков программирования требуют, чтобы индекс массива был целым числом; компилятор должен

сообщить об ошибке, если в качестве индекса массива используется число с плавающей точкой.

Спецификация языка может разрешать определенные преобразования типов,именуемые приведениями (coercion). Например, бинарный арифметический оператор может быть применен либо к паре целых чисел, либо к паре чисел с плавающей точкой. Если такой оператор применен к числу с плавающей точкой и целому числу, то компилятор может выполнить преобразование целого числа в число с плавающей точкой.

В процессе трансляции исходной программы в целевой код компилятор может создавать одно или несколько промежуточных представлений различного вида. Синтаксические деревья являются видом промежуточного представления; обычно они используются в процессе синтаксического и семантического анализа.

После синтаксического и семантического анализа исходной программы многие компиляторы генерируют явное низкоуровневое или машинное промежуточное представление исходной программы, которое можно рассматривать как программу для абстрактной вычислительной машины. Такое промежуточное представление должно обладать двумя важными свойствами: оно должно легко генерироваться и легко транслироваться в целевой машинный язык.

Фаза машинно-независимой оптимизации кода пытается улучшить промежуточный код, чтобы затем получить более качественный целевой код. Обычно «более качественный», «лучший» означает «более быстрый», но могут применяться и другие критерии сравнения, как, например, «более короткий код» или «код, использующий меньшее количество ресурсов».

Имеется большой разброс количества усилий, затрачиваемых различными компиляторами на оптимизацию на этом этапе. Так называемые"оптимизирующие компиляторы" затрачивают на эту фазу достаточно много времени, в то время как другие компиляторы применяют здесь только простые методы оптимизации, которые существенно повышают скорость работы целевой программы, при этом не слишком замедляя процесс компиляции.

Генератор кода получает в качестве входных данных промежуточное представление исходной программы и отображает его в целевой язык. Если целевой язык представляет собой машинный код, для каждой переменной, используемой программой, выбираются соответствующие регистры или ячейки памяти. Затем промежуточные команды транслируются в последовательности машинных команд,выполняющих те же действия. Ключевым моментом генерации кода является аккуратное распределение регистров для хранения переменных.

Фазы связаны с логической организацией компилятора. При реализации работа разных фаз может быть сгруппирована в проходы (pass), которые считывают входной файл и записывают выходной. Например, фазы анализа – лексический анализ, синтаксический анализ, семантический анализ

и генерация промежуточного кода – могут быть объединены в один проход. Оптимизация кода может представлять собой необязательный проход. Затем может быть еще один проход,заключающийся в генерации кода для конкретной целевой машины.

Некоторые наборы компиляторов созданы вокруг тщательно разработанного промежуточного представления, которое позволяет начальной стадии для некоторого языка программирования взаимодействовать с заключительной стадией для определенной целевой машины. При наличии таких наборов можно создавать компиляторы для различных исходных языков и одной целевой машины,комбинируя различные начальные стадии с заключительной стадией для этой целевой машины. Аналогично можно разрабатывать компиляторы для различных целевых машин, комбинируя начальную стадию с заключительными стадиями для различных целевых машин.

2.3. Лексическая спецификация

Обычно все лексемы делятся на классы. Примерами таких классов являются числа (целые, восьмеричные, шестнадцатеричные, действительные и т.д.), идентификаторы, строки. Отдельно выделяются ключевые слова и символы пунктуации (иногда их называют символы-ограничители).

Как правило, ключевые слова – это некоторое конечное подмножество идентификаторов. В некоторых языках (например, ПЛ/1) смысл лексемы может зависеть от ее контекста и невозможно провести лексический анализ в отрыве от синтаксического.

Регулярные выражения представляют собой важный способ записи спецификации шаблонов лексем. Хотя они и не в состоянии выразить все возможные шаблоны, тем не менее регулярные выражения очень эффективны при определении реально используемых для токенов типов шаблонов.

2.4. Вывод в контекстно-свободных грамматиках. Деревья разбора. Неоднозначные грамматики

Как отмечалось раннее, грамматика выводит, или порождает (derive), строки, начиная со стартового символа и неоднократно замещая нетерминалы телами продукций (правыми частями правил) этих нетерминалов (нетерминал должен составлять левую часть правила). Строки токенов, порождаемые из стартового символа, образуют язык, определяемый грамматикой.

Синтаксический анализ, или разбор (parsing), представляет собой выяснение для полученной строки терминалов способа ее вывода из стартового символа грамматики. Если строка не может быть выведена из стартового символа,синтаксический анализатор сообщает об ошибке в строке. Синтаксический анализ –одна из наиболее фундаментальных задач компиляции;

основные методы синтаксического анализа рассматриваются в главе 4. В этой главе для простоты мы начнем с исходных программ наподобие 9-5+2, в которых каждый символ является терминалом; в общем случае исходная программа содержит многосимвольные лексемы, группируемые лексическим анализатором в токены, первые компоненты которых представляют собой терминалы, обрабатываемые синтаксическим анализатором.

Дерево разбора (parse tree) наглядно показывает, как стартовый символ грамматики порождает строку языка. Если нетерминал A имеет продукцию A→XYZ, то дерево разбора может иметь внутренний узел, помеченный как A, с тремя потомками, помеченными слева направо как X, Y и Z.

Формально для данной контекстно-свободной грамматики дерево разбора представляет собой дерево со следующими свойствами.

1. Корень дерева помечен стартовым символом.
2. Каждый лист помечен терминалом или ε.
3. Каждый внутренний узел помечен нетерминалом.
4. Если A является нетерминалом и помечает некоторый внутренний узел, а $X_1,X_2, \ldots,X_n$ – метки его дочерних узлов слева направо, то должна существовать продукция $A \rightarrow X_1X_2\ldots X_n$. Здесь каждое из обозначений X_1, X2, …, X_n представляет собой либо терминальный, либо нетерминальный символ. В качестве частного случая продукции $A \rightarrow \varepsilon$ соответствует A с единственным дочерним узлом ε.

Построение дерева разбора можно сделать совершенно точным при помощи порождений (вывода), рассматривая каждую продукцию как правило для переписывания. Начиная со стартового символа, на каждом шаге переписывания нетерминал замещается телом одной из его продукций. Описанные действия соответствуют нисходящему построению дерева разбора, но точность, достигаемая таким образом,оказывается полезной при рассмотрении восходящего синтаксического анализа. Как мы увидим, восходящий синтаксический анализ связан с классом порождений, известных как правые порождения, когда на каждом шаге переписывается крайний справа нетерминал.

Рассмотрим, например, следующую грамматику с единственным нетерминалом E:

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$

Продукция $E \rightarrow -E$ означает, что если E обозначает выражение, то –E также должно обозначать выражение. Замена одного E на –E может быть описана записью

$$E \Rightarrow -E$$

Она читается как «E порождает –E». Продукция $E \rightarrow (E)$ может быть применена для замены любого экземпляра E в любой строке символов грамматики на (E), например E * E => (E) * E или E * E => E * (E). Можно взять один нетерминал E и многократно применять продукции в произвольном порядке для получения последовательности замещений, например

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

Такая последовательность замен называется выводом (derivation) или порождением –(**id**) из E. Это порождение доказывает, что строка –(**id**) является конкретным примером выражения.

На каждом шаге порождения осуществляется два выбора – мы должны выбрать заменяемый нетерминал, а затем продукцию, у которой данный нетерминал является заголовком.

Чтобы понять, как работают синтаксические анализаторы, рассмотрим порождения, в которых замещаемый нетерминал на каждом шаге выбирается следующим образом.

1. В левых (leftmost) порождениях в каждом предложении все гдавыбирается крайний слева нетерминал. Если $\alpha \Rightarrow \beta$ является шагом, на котором замещается крайний слева нетерминал в α, то это записывается как $\alpha \Rightarrow_{lm} \beta$.
2. В правых (rightmost) порождениях в каждом предложении всегда выбирается крайний справа нетерминал; в этом случае мы пишем $\alpha \Rightarrow_{rm} \beta$.

Каждый левый шаг может быть записан как $wA\gamma \Rightarrow_{lm} w\delta\gamma$, где w состоит только из терминалов, $A \rightarrow \delta$ – примененная продукция, а γ – строка грамматических символов. Чтобы подчеркнуть, что α порождает β путем левого порождения, мы записываем $\alpha \Rightarrow^*_{lm} \beta$. Если $S \Rightarrow^*_{lm} \alpha$, то мы говорим, что α – левосентенциальная форма рассматриваемой грамматики.

Аналогичные определения выполняются и для правых порождений. Правые порождения иногда называются каноническими.

Следует быть предельно внимательным при рассмотрении структуры строки, соответствующей грамматике. Грамматика может иметь более одного дерева разбора для данной строки терминалов. Такая грамматика называется неоднозначной (ambiguous). Чтобы показать неоднозначность грамматики, достаточно найти строку терминалов, которая дает более одного дерева разбора. Поскольку такая строка обычно имеет не единственный смысл, следует разрабатывать однозначные(непротиворечивые) грамматики либо неоднозначные грамматики с дополнительными правилами для разрешения неоднозначностей.

Для большинства синтаксических анализаторов грамматика должна быть однозначной, поскольку, если это не так, мы не в состоянии определить дерево разбора для предложения единственным образом. В некоторых случаях удобно использовать тщательно отобранную неоднозначную грамматику совместно с правилами устранения неоднозначностей (disambiguating rules), которые "отбрасывают" нежелательные деревья разбора, оставляя для каждого предложения по одному дереву.

Иногда для устранения неоднозначности грамматика может быть переписана.

В идеальном мире мы смогли бы дать алгоритм исключения неоднозначности из КС-грамматик. Однако не существует даже алгоритма, способного различить, является ли КС-грамматика неоднозначной. Более того существуют КС-языки, имеющие только неоднозначные КС-грамматики; исключение неоднозначности для них вообще невозможно.

К счастью, положение на практике не настолько мрачное. Для многих конструкций, возникающих в обычных языках программирования, существует техника устранения неоднозначности. Проблема с грамматикой выражений типична, и мы исследуем устранение ее неоднозначности в качестве важной иллюстрации.

Сначала заметим, что есть следующие две причины неоднозначности в грамматике:

не учитываются приоритеты операторов;

последовательность одинаковых операторов может группироваться как слева, так и справа.

Решение проблемы установления приоритетов состоит в том, что вводится несколько разных переменных, каждая из которых представляет выражения, имеющие один и тот же уровень «связывающей мощности».

Хотя порождения не обязательно уникальны, даже если грамматика однозначна, оказывается, что в однозначной грамматике и левые, и правые порождения уникальны.

2.5. Базовые блоки и граф потока

В трехадресном коде в правой части команды имеется не более одного оператора, то есть не допускаются никакие встроенные арифметические выражения. Таким образом, выражение на исходном языке наподобие x+y*z может быть транслировано в трехадресные команды

$$t_1 = y * z$$
$$t_2 = x + t_1$$

Здесь вводится представление промежуточного кода в виде графа,полезное при рассмотрении генерации кода (даже если алгоритм генерации кода не строит граф явным образом). Генерация кода выигрывает от знания контекста.

Представление строится следующим образом.

Промежуточный код разделяется на базовые блоки (basic blocks), представляющие собой максимальные последовательности следующих друг за другом трехадресных команд, обладающие приведенными ниже свойствами:

поток управления может входить в базовый блок только через первую команду блока, то есть переходы в средину блока отсутствуют;

управление покидает блок без останова или ветвления, за исключением, возможно, в последней команде блока.

Базовые блоки становятся узлами графа потока (flow graph), ребра которого указывают порядок следования блоков.

Представление о том, что управление, достигнув начала базового блока, обязательно дойдет до его конца, требует пояснений. Имеется множество причин для прерываний, явно не отражаемых в коде, которые могут заставить управление покинуть блок, возможно, навсегда. Например, команда наподобие x=y/z кажется не влияющей на поток управления, но если z равно 0, то она может привести к завершению программы. Мы не будем беспокоиться о таких возможностях. Причина этого в следующем. Цель построения базовых блоков – оптимизация кода. В общем случае при генерации исключения либо оно будет обработано и управление вернется к команде, вызвавшей его, как если бы никакого исключения не было, либо программа завершится с кодом ошибки. В последнем случае оптимизация кода не имеет значения, поскольку в любом случае программа не даст требуемого результата.

Наша первая задача состоит в разбиении последовательности трехадресных команд на базовые блоки. Мы начинаем новый базовый блок с первой команды и добавляем команды до тех пор, пока не встретим условный или безусловный переход или метку у следующей команды. При отсутствии переходов и меток управление последовательно проходит от одной команды к другой.

После того как программа разбита на базовые блоки, поток управления между ними представляется в виде графа. Узлами графа потока являются базовые блоки. Ребро из блока B в блок C идет тогда и только тогда, когда первая команда блока C может следовать непосредственно за последней командой блока B. Имеются две ситуации, когда могут существовать такие ребра.

1. Существует условный или безусловный переход от конца блока B к началу блока C.
2. C следует непосредственно за Б в исходном порядке трехадресных команд,а блок B не заканчивается безусловным переходом.

Мы говорим, что B – предшественник C, а C – преемник B.

Зачастую к графу добавляются два узла, именуемые входом и выходом и несоответствующие выполнимым промежуточным командам. Существует ребро от входа к первому выполнимому узлу графа, то есть к базовому блоку, который начинается с первой команды промежуточного кода. Если последняя команда программы не является безусловным переходом, то выходу предшествует блок, содержащий эту последнюю команду программы. В противном случае таковым предшествующим блоком является любой базовый блок, имеющий переход к коду, не являющемуся частью программы.

3. СПОСОБЫ ЗАДАНИЯ ГРАММАТИК

3.1. Форма Бэкуса–Наура

Форма Бэкуса–Наура (сокр. БНФ, Бэкуса–Наура форма) – формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик. Существует расширенная форма Бэкуса–Наура, отличающаяся лишь более ёмкими конструкциями.

Используется для описания синтаксиса языков программирования, данных, протоколов (например, в документах RFC) и т.д. (причём как грамматики, так и регулярной лексики, поскольку регулярные грамматики являются подмножеством контекстно-свободных).

БНФ-конструкция определяет конечное число символов (нетерминалов). Кроме того, она определяет правила замены символа на какую-то последовательность букв (терминалов) и символов. Процесс получения цепочки букв можно определить поэтапно: изначально имеется один символ (символы обычно заключаются в угловые скобки, а их название не несёт никакой информации). Затем этот символ заменяется на некоторую последовательность букв и символов, согласно одному из правил. Затем процесс повторяется (на каждом шаге один из символов заменяется на последовательность, согласно правилу). В конце концов, получается цепочка, состоящая из букв и не содержащая символов. Это означает, что полученная цепочка может быть выведена из начального символа.

БНФ-конструкция состоит из нескольких предложений вида

<определяемый символ> ::= <посл.1> | <посл.2> | … | <посл.n>

Эти конструкции описывают правила. Такое правило означает, что символ <определяемый символ> может заменяться на одну из последовательностей <посл.n>. Знак определения обычно выглядит как ::= или →, но возможны и другие варианты.

Некоторые специальные символы, как например <пусто>, означают какую-то последовательность (в данном случае – пустую).

3.2. Расширенная форма Бэкуса–Наура

Расширенная форма Бэкуса–Наура (расширенная Бэкуса–Наура форма (РБНФ)) (англ. Extended Backus–Naur Form (EBNF)) – формальная система определения синтаксиса, в которой одни синтаксические категории последовательно определяются через другие. Используется для описания контекстно-свободных формальных грамматик. Предложена Никлаусом Виртом. Является расширенной переработкой форм Бэкуса–Наура, отличается

от БНФ более «ёмкими» конструкциями, позволяющими при той же выразительной способности упростить и сократить в объёме описание.

Тем не менее, используется множество различных вариантов РБНФ. Международная организация по стандартизации приняла стандарт РБНФ: ISO/IEC 14977.

Как и в БНФ, описание грамматики в РБНФ представляет собой набор правил, определяющих отношения между терминальными символами (терминалами) и нетерминальными символами (нетерминалами).

Терминальные символы – это минимальные элементы грамматики, не имеющие собственной грамматической структуры. В РБНФ терминальные символы – это либо предопределённые идентификаторы (имена, считающиеся заданными для данного описания грамматики), либо цепочки – последовательности символов в кавычках или апострофах.

Нетерминальные символы – это элементы грамматики, имеющие собственные имена и структуру. Каждый нетерминальный символ состоит из одного или более терминальных и/или нетерминальных символов, сочетание которых определяется правилами грамматики. В РБНФ каждый нетерминальный символ имеет имя, которое представляет собой строку символов.

Правило в РБНФ имеет вид:

идентификатор = выражение.

где идентификатор – имя нетерминального символа, а выражение – соответствующая правилам РБНФ комбинация терминальных и нетерминальных символов и специальных знаков. Точка в конце – специальный символ, указывающий на завершение правила.

Семантика правила РБНФ – нетерминальный символ, заданный идентификатором слева от знака «равно», представляет собой определяемую выражением комбинацию терминальных и нетерминальных символов.

Полное описание грамматики представляет собой набор правил, который последовательно определяет все нетерминальные символы грамматики так, что каждый нетерминальный символ может быть сведён к комбинации терминальных символов путём последовательного (рекурсивного) применения правил. В определении РБНФ нет никаких специальных предписаний относительно порядка записи правил, хотя такие предписания могут вводиться при использовании РБНФ программными средствами, обеспечивающими автоматическую генерацию программ синтаксического разбора по описанию грамматики.

Набор возможных конструкций РБНФ очень невелик. Это конкатенация, выбор, условное вхождение и повторение.

Конкатенация. Определяется символом "," (запятая). Правило вида A = B,C. обозначает, что нетерминал A состоит из двух символов – B и C. Элементы конкатенации называют ещё синтаксическими факторами, или просто факторами. В данном примере B и C – синтаксические факторы. В некоторых вариантах РБПФ допустимо запятую не ставить.

Выбор. Обозначается вертикальной чертой. Правило вида A = B|C|D. обозначает, что нетерминал A может состоять либо из B, либо из C, либо из D. Элементы выбора называют ещё синтаксическими термами, или просто термами. В данном примере B, C, D – синтаксические термы.

Условное вхождение. Квадратные скобки выделяют необязательный элемент выражения, который может присутствовать, а может и отсутствовать. Правило вида A = [B]. обозначает, что нетерминал A либо является пустым, либо состоит из символа B.

Повторение. Фигурные скобки обозначают конкатенацию любого числа (включая нуль) записанных в ней элементов. Правило вида A = {B}. обозначает, что A – либо пустой, либо представляет собой конкатенацию любого числа символов B (то есть A – это либо пустой элемент, либо B, либо BB, либо BBB и так далее). Если требуется, чтобы A представлял собой либо B, либо произвольное число B, но не мог быть пустым, используется запись A = B{B}.

Помимо основных операций, в РБНФ могут использоваться обычные круглые скобки. Они применяются для группировки элементов при формировании сложных выражений. Например, правило A = (B|C)(D|E). обозначает, что A состоит из двух символов, первым из которых является либо B, либо C, вторым – либо D, либо E, то есть A может быть одной из цепочек BD, BE, CD, CE.

В некоторых работах встречаются модифицированные варианты синтаксиса РБНФ.

Можно встретить использование в правилах символа «::=» вместо «=» (по аналогии с БНФ).

Иногда конкатенация в выражениях обозначается не простым следованием символов друг за другом, а с помощью запятой. В таком случае несколько слов, написанных через пробелы, следует понимать как одно многословное имя нетерминального символа.

3.3. Итерационная форма

Для получения более компактных описаний синтаксиса применяют итерационную форму описания. Такая форма предполагает введение специальной операции, которая называется итерацией и обозначается парой фигурных скобок со звездочкой. Итерация вида $\{a\}^*$ определяется как множество, включающее цепочки всевозможной длины, построенные с использованием символа a, и пустую цепочку.

$\{a\}^* = \{\varepsilon, a, aa, aaa, aaaa,...\}$.

Используя итерацию для описания множества цепочек, задаваемых символическими правилами, для списка получаем:

$L \rightarrow E \{E\}^*$.

Например, описание множества цепочек, каждая из которых должна начинаться знаком # и может состоять из произвольного числа букв x и y, может быть представлено в итерационной форме так:

$I \rightarrow \#\{x \mid y\}^*$.

В итерационных формах описания наряду с итерационными скобками часто применяют квадратные скобки для указания того, что цепочка, заключенная в них, может быть опущена. С помощью таких скобок правила:

$A \rightarrow xAyBz$ и $A \rightarrow xBz$

могут быть записаны так:

$A \rightarrow x[Ay]Bz$.

3.4. Синтаксические диаграммы

Синтаксическая диаграмма – это направленный граф с одним входным ребром и одним выходным ребром и помеченными вершинами. Синтаксическая диаграмма задаёт язык. Цепочка пометок при вершинах на любом пути от входного ребра к выходному – это цепочка языка, задаваемого синтаксической диаграммой. Между входным/выходным рёбрами находятся блоки двух видов: «круг» – определяет базовое (первичное) понятие; и «прямоугольник» – определяет вторичное понятие, которое определённо (то есть метапеременная). Поэтому можно считать, что синтаксическая диаграмма – это одна из форм порождающей грамматики автоматных языков. Синтаксические диаграммы и конечные автоматы имеют тесную связь: любой автоматный язык задаётся синтаксической диаграммой и обратно, по любой синтаксической диаграмме можно построить конечный автомат (в общем случае недетерминированный), распознающий тот же язык, который задаёт диаграмма.

Построив по синтаксической диаграмме соответствующий распознающий конечный автомат, можно затем реализовать этот автомат либо аппаратно, либо программно.

Таким образом, синтаксические диаграммы могут служить не только для порождения, но и для распознавания автоматных языков.

3.5. Способ описания грамматики для языков C, C++ и подобных

Такой способ использовался в книге Б.Керниган, Д.Ритчи «Язык программирования C»

Нетерминалы записываются курсивом.

Терминалы записываются прямым шрифтом.

Левая часть правила записывается отдельной строкой с отступом влево.

Альтернативные части правил выписываются в столбик по одному в строке или помечаются словами «one of».

Необязательные части обозначаются нижним индексом «opt».

Средства для обозначения повторений отсутствуют, поэтому в описаниях часто используется рекурсия.

4. ПОСТРОЕНИЕ ГРАММАТИК. ЭКВИВАЛЕНТНЫЕ ПРЕОБРАЗОВАНИЯ КОНТЕКСТНО-СВОБОДНЫХ ГРАММАТИК

4.1. Рекомендации по построению грамматик

Грамматикой можно описать большую часть (но не весь) синтаксиса языков программирования. Например, требование объявления идентификаторов до их использования не может быть описано контекстно-свободной грамматикой. Значит, последовательности токенов, принимаемые синтаксическим анализатором, образуют надмножество языка программирования; последующие фазы компилятора должны анализировать выход синтаксического анализатора, чтобы обеспечить его соответствие правилам, которые не проверяются синтаксическим анализатором.

Все, что может быть описано при помощи регулярного выражения, может быть также описано и грамматикой. Закономерен вопрос: почему для определения лексического синтаксиса языка используются регулярные выражения? На то есть несколько причин.

Разделение синтаксической структуры языка на лексическую и не лексическую части представляет собой удобный способ разбиения начальной стадии компиляции на два модуля меньшего размера, что упрощает работу с ними.

Лексические правила языка часто очень просты, и для их описания не требуется такое мощное средство, как грамматика.

Лексические правила языка часто очень просты, и для их описания не требуется такое мощное средство, как грамматика.

Автоматически создаваемые на основе регулярных выражений лексические анализаторы более эффективны, чем лексические анализаторы, построенные на основе грамматики.

Четких правил, что именно следует отнести к лексическим правилам, а что – к синтаксическим, нет. Регулярные выражения в большей степени подходят для описания структуры таких конструкций, как идентификаторы, константы, ключевые слова и пробельные символы. Грамматики же более приспособлены для описания вложенных структур, таких как сбалансированные скобки, пары begin-end, соответствующие друг другу if-then-else, и т.п. Эти вложенные структуры не могут быть описаны регулярными выражениями.

В зависимости от применяемого при синтаксическом анализе метода, от грамматик может требоваться выполнение различных условий. К общим требованиям относится однозначность грамматики, отсутствие в ней непродуктивных (из которых нельзя вывести цепочку состоящую только из терминальных символов) и недостижимых (которые не могут встречаться в сентенциальных формах) символов.

Общий подход к построению грамматики состоит в следующем. Каждому нетерминальному символу ставится в соответствие некоторый фиксированный конечный набор синтаксических конструкций, а стартовому символу сопоставляется весь порождаемый язык. Далее для каждого нетерминального символа записываются правила показывающие, как он может быть представлен в виде цепочки терминальных символов и синтаксических конструкций, соответствующих нетерминальным.

4.2. Примеры грамматик

Если не установлено иное, для краткости будем пользоваться следующими соглашениями:

нетерминальные символы обозначаются одиночными заглавными буквами латинского алфавита;

терминальные символы обозначаются знаками (+, – и тому подобными) либо строчными буквами латинского алфавита (словами из таких букв) или цепочками символов в кавычках.

правила группируются по совпадающим левым частям, то есть будут записываться в виде $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$.

стартовый символ грамматики определяется как нетерминальный символ в левой части первого указанного правила.

Таким образом, грамматика полностью описывается набором правил.

Пример Грамматика для идентификаторов

I → L | I L | I D

L → «a» | «b» | … | «z» | «A» | «B» | … | «Z» | «_»

D → «0» | «1» | «2» | «3» | «4» | «5» | «6» | «7» | «8» | «9»

Пример Грамматика для целочисленных литералов

I → D | ID

D → «0» | «1» | «2» | «3» | «4» | «5» | «6» | «7» | «8» | «9»

Пример Грамматика для числовых литералов

N → S I E | S . I E | S I F E

S → + | – | ε

E → +I | –I | ε

I → D | I D

F → . | F D

D → «0» | «1» | «2» | «3» | «4» | «5» | «6» | «7» | «8» | «9»

Пример Грамматика для арифметических выражений, содержащих операции унарный минус, сложения и умножения, а также скобки, числа и переменные (идентификаторы).

E → –E | E + T | T

T → T * F | F

F → (E) | **num** | **id**

Пример Объявление переменных, массивов и указателей типа int или float с автоматической длительностью хранения (указатели на массивы этой грамматикой не порождаются)

Объявление:

D → B L ;

Базовый тип:

B → «float» | «int»

Список деклараторов:

L → P **id** A W | P **id** V | L, P **id** A W | L, P **id** V

Указатель:

P → ε | * P

Массив:

A → ε | «[» «]» | A «[» **num** «]»

Инициализация массива:

W → ε | = «{» N «}»

Инициализация переменной

V → ε | = E

Список выражений

N → E | N, E

Выражение:

E → –E | E + T | T

Слагаемое:

T → T * F | F

Множитель:

F → (E) | **num** | **id**

Пример Последовательность операторов-выражений с присваиваниями (без поддержки арифметики операторов)

Последовательность:

S → A | S A

Оператор выражение-присваивания:

A → E ; | D = A

Разыменование:

D → * D | I

Обращение по индексу:

I → I «[» E «]» | (D) | **id**

Пример Инструкция ветвления (здесь S – символ соответствующий объединению языков всех инструкций, включая последовательность, заключенную в фигурные скобки; E – выражение)

C → «if» «(» E «)» S | «if» «(» E «)» S «else» S

Пример Инструкция повторения (здесь S – символ соответствующий объединению языков всех инструкций, включая последовательность, заключенную в фигурные скобки; E – выражение)

L → «while» «(» E «)» S

4.3. Нормальная форма Хомского

Каждый КС-язык (без ε) порождается грамматикой, все продукции которой имеют форму $A \to BC$ или $A \to a$, где A, B и C – нетерминалы, a – терминал. Эта форма называется нормальной формой Хомского. Для ее получения нужно несколько предварительных преобразований, имеющих самостоятельное значение.

1. Удалить бесполезные символы, т.е. нетерминалы или терминалы, которые не встречаются в порождениях терминальных цепочек из стартового символа.
2. Удалить ε-продукции, т.е. продукции вида $A \to \varepsilon$ для некоторого нетерминала A.
3. Удалить цепные продукции вида $A \to B$ с нетерминалами A и B.

Символ X называется полезным в грамматике $G = (V, T, P, S)$, если существует некоторое порождение вида $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$, где $w \in T^*$. Отметим, что X может быть как нетерминалом, так и терминалом, а выводимая цепочка $\alpha X \beta$ – первой или последней в порождении. Если символ X не является полезным, то называется бесполезным. Очевидно, что исключение бесполезных символов из грамматики не изменяет порождаемого языка, поэтому все бесполезные символы можно обнаружить и удалить.

Наш подход к удалению бесполезных символов начинается с определения двух свойств, присущих полезным символам.

Символ X называется порождающим, если $X \Rightarrow^* w$ для некоторой терминальной цепочки w. Заметим, что каждый терминал является порождающим, поскольку w может быть этим терминалом, порождаемым за 0 шагов.

Символ X называется достижимым, если существует порождение $S \Rightarrow^* \alpha X \beta$ для некоторых α и β.

Естественно, что полезный символ является одновременно и порождающим, и достижимым. Если сначала удалить из грамматики непорождающие символы, а затем недостижимые, то, останутся только полезные.

Теорема Пусть $G = (V, T, P, S)$ – КС-грамматика, и $L(G) \neq \emptyset$, т.е. G порождает хотя бы одну цепочку. Пусть $G_1 = (V_1, T_1, P_1, S)$ – грамматика, полученная с помощью следующих двух шагов.

1. Вначале удаляются непорождающие символы и все продукции, содержащие один или несколько таких символов. Пусть $G_2 = (V_2, T_2, P_2, S)$ – полученная в результате грамматика. Заметим, что S должен быть порождающим, так как по предположению L(G) содержит хотя бы одну цепочку, поэтому S не удаляется.
2. Затем удаляются все символы, недостижимые в G_2.

Тогда G_1 не имеет бесполезных символов, и $L(G_1) = L(G)$.

Рассмотрим, каким образом вычисляются множества порождающих и достижимых символов грамматики. В алгоритме, используемом в обеих задачах, делается максимум возможного, чтобы обнаружить символы этих

двух типов. Докажем, что если правильное индуктивное построение указанных множеств не позволяет обнаружить, что символ является порождающим или достижимым, то он не является символом соответствующего типа.

Базис. Каждый символ из T, очевидно, является порождающим; он порождает сам себя.

Индукция. Предположим, есть продукция $A \to \alpha$, и известно, что каждый символ в α является порождающим. Тогда A – порождающий. Заметим, что это правило включает и случай, когда $\alpha = \varepsilon$; все нетерминальные символы, имеющие ε в качестве тела продукции, являются порождающими.

Теорема Вышеприведенный алгоритм находит все порождающие символы грамматики G и только их.

Теперь рассмотрим индуктивный алгоритм, с помощью которого находится множество достижимых символов грамматики G = (V, T, P, S). Можно доказать, что если символ не добавляется к множеству достижимых символов путем максимально возможного обнаружения, то он не является достижимым.

Базис. Очевидно, что S действительно достижим.

Индукция. Пусть обнаружено, что некоторый нетерминал A достижим. Тогда для всех продукций с заголовком A все символы тел этих продукций также достижимы.

Теорема Вышеприведенный алгоритм находит все достижимые символы грамматики G, и только их.

Покажем теперь, что ε-продукции, хотя и удобны в задачах построения грамматик, не являются существенными. Конечно же, без продукции с телом ε невозможно породить пустую цепочку как элемент языка. Таким образом, в действительности доказывается, что если L задается КС-грамматикой, то $L \setminus \{\varepsilon\}$ имеет КС-грамматику без ε-продукций.

Начнем с обнаружения «ε-порождающих» нетерминалов. Нетерминал A называется ε -порождающей, если $A \Rightarrow^* \varepsilon$. Если A – ε-порождающая, то где бы в продукциях она ни встречалась, например в $B \to CAD$, из нее можно (но не обязательно) вывести ε. Продукция с такой переменной имеет еще одну версию без этой переменной в теле ($B \to CD$).

Эта версия соответствует тому, что ε-порождающий нетерминал использована для вывода ε. Используя версию $B \to CAD$, мы не разрешаем из A выводить ε. Это не создает проблем, так как далее мы просто удалим все продукции с телом ε, предохраняя каждый нетерминал от порождения ε.

Пусть G = (V, T, P, S) – КС-грамматика. Все ε-порождающие символы G можно найти с помощью следующего алгоритма. Далее будет показано, что других ε-порождающих символов, кроме найденных алгоритмом, нет.

Базис. Если $A \to \varepsilon$ – продукция в G, то A – ε-порождающая.

Индукция. Если в G есть продукция $B \to C_1 C_2 ... C_k$, где каждый символ C_i является ε-порождающим, то B – ε-порождающая. Отметим, что для того, чтобы C_i был ε-порождающим, он должен быть нетерминалом, поэтому нам

нужно рассматривать продукции, тела которых содержат только нетерминалы.

Теорема В любой грамматике ε-порождающими являются только переменные, найденные вышеприведенным алгоритмом.

Пример Рассмотрим следующую грамматику.

S → AB

A → aAA | ε

B → bBB | ε

Сначала найдем ε-порождающие символы. A и B непосредственно ε-порождающие, так как имеют продукции с ε в качестве тела. Тогда и S ε-порождающий, поскольку тело продукции S → AB состоит только из ε-порождающих символов. Таким образом, все три нетерминала являются ε-порождающими.

Построим теперь продукции грамматики G_1. Сначала рассмотрим S → AB. Все символы тела являются ε-порождающими, поэтому есть 4 способа выбрать присутствие или отсутствие A и B. Однако нам нельзя удалять все символы одновременно, поэтому получаем следующие три продукции.

S → AB | A | B

Далее рассмотрим продукцию A → aAA. Вторую и третью позиции занимают ε-порождающие символы, поэтому снова есть 4 варианта их присутствия или отсутствия. Все они допустимы, поскольку в любом из них остается терминал a. Два из них совпадают,

поэтому в грамматике G1 будут следующие три продукции.

A → aAA | aA | a

Аналогично, продукция для B приводит к следующим продукциям в G_1.

B → bBB | bB | b

Обе ε-продукции из G не вносят в G_1 ничего. Таким образом, следующие продукции образуют G_1.

S → AB | A | B

A → aAA | aA | a

B → bBB | bB | b

Теорема Если грамматика G1 построена по грамматике G с помощью описанной выше конструкции удаления ε-продукций, то $L(G_1) = L(G) \setminus \{\varepsilon\}$.

Цепная продукция – это продукция вида A → B, где и A, и B являются нетерминалами.

Цепные продукции могут усложнять некоторые доказательства и создавать излишние шаги в порождениях, которые по техническим соображениям там совсем не нужны.

Особую сложность представляет случай, когда в грамматике есть цикл из цепных продукций, вроде A → B, B → C и C → A. Техника, гарантирующая результат, включает первоначальное нахождение всех пар переменных A и B, для которых $A \Rightarrow^* B$ получается с использованием последовательности лишь

цепных продукций. Заметим, что $A \Rightarrow^* B$ возможно и без использования цепных продукций, например, с помощью продукций $A \to BC$ и $C \to \varepsilon$.

Определив все подобные пары, любую последовательность шагов порождения, в которой $A \Rightarrow B1 \Rightarrow B\,2 \Rightarrow \ldots \Rightarrow Bn \Rightarrow \alpha$ с нецепной продукцией $Bn \to \alpha$, можно заменить продукцией $A \to \alpha$. Однако вначале рассмотрим индуктивное построение пар (A, B), для которых $A \Rightarrow^* B$ получается с использованием лишь цепных продукций. Назовем такую пару цепной парой (unit pair).

Базис. (A, A) является цепной парой для любой переменной, т.е. $A \Rightarrow^* A$ за нуль шагов.

Индукция. Предположим, что пара (A, B) определена как цепная, и $B \to C$ – продукция с переменной C. Тогда (A, C) – цепная пара.

Способ построения пар теперь очевиден. Нетрудно доказать, что предложенный алгоритм обеспечивает порождение всех нужных пар. Зная эти пары, можно удалить цепные продукции из грамматики и показать эквивалентность исходной и полученной грамматик.

Теорема Приведенный выше алгоритм находит все цепные пары грамматики G, и только их.

Для удаления цепных продукций по КС-грамматике G = (V, T, P, S) построим КС-грамматику $G_1 = (V_1, T, P_1, S)$ следующим образом.

1. Найдем все цепные пары грамматики G.
2. Для каждой пары (A, B) добавим к P_1 все продукции $A \to \alpha$, где $B \to \alpha$ – нецепная продукция из P. Заметим, что при A = B все нецепные продукции для B из P просто добавляются к P_1.

Теорема Если грамматика G_1 построена по грамматике G с помощью алгоритма удаления цепных продукций, описанного выше, то $L(G_1) = L(G)$.

Теорема Если G – КС-грамматика, порождающая язык, в котором есть хотя бы одна непустая цепочка, то существует другая грамматика G_1, не имеющая бесполезных символов, ε-продукций и цепных продукций, у которой $L(G_1) = L(G) \setminus \{\varepsilon\}$.

Для каждого непустого КС-языка, не включающего ε, существует грамматика G, все продукции которой имеют одну из следующих двух форм.

1. $A \to BC$, где A, B и C – нетерминальные.
2. $A \to a$, где A – нетерминал, a – терминал.

Кроме того, G не имеет бесполезных символов. Такая форма грамматик называется нормальной формой Хомского, или НФХ, а грамматики в такой форме – НФХ-грамматиками.

Для приведения грамматики к НФХ начнем с ее формы, свободной от бесполезных символов, цепных и ε-продукций. Каждая продукция такой грамматики либо имеет вид $A \to a$, допустимый НФХ, либо имеет тело длиной не менее 2. Нужно выполнить следующие преобразования:

1. устроить так, чтобы все тела длины 2 и более состояли только из нетерминалов;

2. разбить тела длины 3 и более на группу продукций, тело каждой из которых состоит из двух нетреминалов.

Конструкция для 1 следующая. Для каждого терминала a, встречающегося в продукции длины 2 и более, создаем новый нетерминал, скажем, A. Этот нетерминал имеет единственную продукцию A → a. Используем переменную A вместо a везде в телах продукций длины 2 и более. Теперь в теле каждой продукции либо одиночный терминал, либо как минимум два нетерминала и нет терминалов.

Для шага 2 нужно разбить каждую продукцию вида $A \to B_1B_2\ldots B_k$, где $k \geq 3$, на группу продукций с двумя переменными в каждом теле. Введем $k - 2$ новых переменных $C_1, C_2, \ldots, C_{k-2}$ и заменим исходную продукцию на $k - 1$ следующих продукций.

$$A \to B_1C_1,\ C_1 \to B_2C_2,\ \ldots,\ C_{k-3} \to B_{k-2}C_{k-2},\ C_{k-2} \to B_{k-1}B_k$$

Теорема Если G – КС-грамматика, язык которой содержит хотя бы одну непустую цепочку, то существует НФХ-грамматика G_1, причем $L(G1) = L(G) \setminus \{\varepsilon\}$.

4.4. Исключение леворекурсивных правил

Грамматика является леворекурсивной (left recursive), если в ней имеется нетерминал A, такой, что существует порождение $A \Rightarrow^+ A\alpha$ для некоторой строки a. Методы нисходящего разбора не в состоянии работать с леворекурсивными грамматиками, поэтому требуется преобразование грамматики, которое устранило бы из нее левую рекурсию. Сначала рассмотрим непосредственнуюлевую рекурсию (immediate left recursion), при которой существует продукция вида $A \to A\alpha$. Затем будет рассмотрен общий случай.

Непосредственная левая рекурсия может быть устранена при помощи следующего метода, который работает для любого количества A-продукций. Вначале сгруппируем A-продукции следующим образом

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Здесь ни одно β_i не начинается с A. Затем заменим эти A-продукции следующим образом:

$A \to \beta_1\ A' \mid \beta_2\ A' \mid \ldots \mid \beta_n\ A'$

$A' \to \alpha_1\ A' \mid \alpha_2\ A' \mid \ldots \mid \alpha_m\ A' \mid \varepsilon$

Нетерминал A порождает те же строки, что и ранее, но без левой рекурсии. Эта процедура устраняет все левые рекурсии из продукций для A и A' (при условии,что ни одна строка α_i не является ε), но не устраняет левую рекурсию, вызванную двумя или более шагами порождения.

Приведенный ниже алгоритм, систематически удаляет из грамматики левую рекурсию. Он гарантированно работает с грамматиками, не имеющими циклов (порождений типа $A =>^+ A$) и ε-продукций.

Алгоритм Устранение левой рекурсии

Вход: грамматика G без циклов и ε-продукций.

Выход: эквивалентная грамматика без левой рекурсии.

Расположить нетерминалы в некотором порядке A1, A2, …, An.

```
for ( каждое i от 1 до n ) {
    for ( каждое j от 1 до i – 1 ) {
        заменить каждую продукцию вида Ai → Ajγ продукциями
        Ai → δ1γ | δ2γ | … | δkγ, где Aj -> δ1 | δ2 | … | δk –
все
        текущие Aj-продукции
    }
    устранить непосредственную левую рекурсию среди A{ -
продукций
}
```

Обратите внимание, что получающаяся грамматика без левых рекурсий может иметь ε-продукции.

4.5. Левая факторизация

Левая факторизация (left factoring) представляет собой преобразование грамматики в пригодную для предиктивного, или нисходящего, синтаксического анализа. Когда не ясно, какая из двух альтернативных продукций должна использоваться для нетерминала A, A-продукции можно переписать так, чтобы отложить принятие решения до тех пор, пока из входного потока не будет прочитано достаточно символов для правильного выбора.

Алгоритм Левая факторизация грамматики

Вход: грамматика G.

Выход: эквивалентная левофакторизованная грамматика.

Метод: для каждого нетерминала A находим самый длинный префикс a, общий для двух или большего числа альтернатив. Если $\alpha \neq \varepsilon$, т.е. имеется нетривиальный общий префикс, заменим все продукции $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma$, где γ представляет все альтернативы, не начинающиеся с α, продукциями

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

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

Здесь A’ – новый нетерминал. Выполняем это преобразование до тех пор, пока никакие две альтернативы нетерминала не будут иметь общий префикс.

5. КОНЕЧНЫЕ АВТОМАТЫ

5.1. Понятие конечного автомата. Определение недетерминированного конечного автомата

Конечный автомат (КА) в теории алгоритмов – математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. Является частным случаем абстрактного дискретного автомата, число возможных внутренних состояний которого конечно.

При работе на вход КА последовательно поступают входные воздействия, а на выходе КА формирует выходные сигналы. Обычно под входными воздействиями принимают подачу на вход автомата символов одного алфавита, а на выход КА в процессе работы выдаёт символы в общем случае другого, возможно даже не пересекающегося со входным, алфавита.

Помимо конечных автоматов существуют и бесконечные дискретные автоматы – автоматы с бесконечным числом внутренних состояний.

Переход из одного внутреннего состояния КА в другое может происходить не только от внешнего воздействия, но и самопроизвольно.

Различают детерминированные КА – автоматы, в которых следующее состояние однозначно определяется текущим состоянием и входным символом и выход зависит только от текущего состояния и текущего входа, и недетерминированные КА, следующее состояние у которых в общем случае не определено и, соответственно, не определён выходной сигнал. Если переход в последующие состояния происходит с некоторыми вероятностями, то такой КА называют вероятностным КА.

Примерами физической реализации КА могут служить любые цифровые системы, например, компьютеры или некоторые логические узлы компьютеров с памятью – триггеры и другие устройства. Комбинационная последовательная логика не может являться КА, так как не имеет внутренних состояний (не имеет памяти).

С абстрактной точки зрения КА изучается разделом дискретной математики – теория конечных автоматов.

Теория конечных автоматов практически широко используется, например, в синтаксических и лексических анализаторах, тестировании программного обеспечения на основе моделей.

Недетерминированный конечный автомат (НКА, англ. nondeterministic finite automaton, NFA) – это детерминированный конечный автомат (ДКА, англ. deterministic finite automaton, DFA), который не выполняет следующие условия:

любой его переход единственным образом определяется по текущему состоянию и входному символу

чтение входного символа требуется для каждого изменения состояния.

В частности, любой ДКА является также НКА.

Используя алгоритм конструкции подмножеств, любой НКА можно преобразовать в эквивалентный ДКА, то есть ДКА, распознающий тот же самый формальный язык. Подобно ДКА, НКА распознаёт только регулярные языки.

НКА предложили в 1959 году Михаэль О. Рабин и Дана Скотт, которые показали его эквивалентность ДКА. НКА используется в реализации регулярных выражений – построение Томпсона является алгоритмом для преобразования регулярного выражения в НКА, который может эффективно распознавать шаблон строк. Обратно, алгоритм Клини можно использовать для преобразования НКА в регулярное выражение, размер которого в общем случае экспоненциально зависит от размера автомата.

НКА обобщён многими путями, например: недетерминированным конечным автоматом с ε-переходами, преобразователями с конечным числом состояний, автоматами с магазинной памятью, альтернирующими автоматами, ω-автоматами и вероятностными автоматами. Кроме ДКА известны другие специальные случаи НКА – однозначные конечные автоматы (англ. unambiguous finite automata, UFA) и самопроверочные конечные автоматы (англ. self-verifying finite automata, SVFA).

Есть несколько неформальных эквивалентных описаний:

НКА, подобно ДКА, принимает строку входных символов. Для каждого входного символа он переходит в новое состояние, пока не обработает все входные символы. На каждом шаге автомат произвольным образом выбирает один из возможных переходов. Если существует «удачный проход», то есть некоторая последовательность выборов, приводящая к конечному состоянию после полной выборки входной строки, то строка принимается. Если же нет последовательности, которая после обработки всей входной строки приводит автомат в конечное состояние, то входная строка отвергается.

Пусть опять НКА принимает строку входных символов, один символ за другим. На каждом шаге, где два или более перехода оказываются допустимыми, автомат «клонирует» себя на нужное число копий, каждая из которых осуществляет различные переходы. Если никакой из переходов не может быть осуществлён, текущая копия является тупиком и «умирает». Если после выборки всех символов из входной строки какая-либо из копий переходит в конечное состояние, входная строка принимается, в противном случае – отвергается.

НКА формально представляется как 5-кортеж $(Q, \Sigma, \Delta, q_0, F)$, состоящий из:

конечного множества состояний Q.

конечного множества входных символов Σ.

функции переходов $\Delta : Q \times \Sigma \to P(Q)$. (P(X) – набор всех подмножеств множества X.)

начального состояния $q_0 \in Q$.

множества состояний F распознаваемых как конечные (заключительные) состояния $F \subseteq Q$.

Если дан НКА $M = (Q, \Sigma, \Delta, q_0, F)$, он распознаёт язык, который обозначается как L (M) и определяется как множество всех строк над алфавитом Σ, принимаемых автоматом M.

В общих чертах согласно неформальным объяснениям выше, существует несколько эквивалентных формальных определений строки $w = a_1a_2\ldots a_n$, принимаемых автоматом M:

w принимается, если существует последовательность состояний r_0 , r_1 , …, r_n в Q такая, что

$r_0 = q_0$

$r_{i+1} \in \Delta(r_i, a_{i+1})$, для $i = 0, \ldots, n - 1$

$r_n \in F$.

Другими словами. Первое условие гласит, что машина начинает работу из состояния q_0. Второе условие гласит, что для каждого символа строки w машина переходит из состояния в состояние согласно функции переходов Δ. Последнее условие гласит, что машина принимает строку w, если входная строка w приводит машину к завершению в конечном состоянии. Чтобы строка w была принята автоматом M, не требуется, чтобы любая последовательность состояний завершается в конечном состоянии, достаточно, чтобы в такое состояние приводила одна последовательность. В противном случае, то есть, если невозможно перейти из q_0 в состояние из F, следуя w, говорят, что автомат отвергает строку. Множество строк, которые автомат M принимает, является языком, распознаваемым автоматом M, и этот язык обозначается как L(M).

Определение автомата выше использует одно начальное состояние, что не является обязательным условием. Иногда НКА определяется с множеством начальных состояний. Существует простое построение[en], которое переносит НКА с несколькими начальными состояниями в НКА с одним начальным состоянием.

Детерминированный конечный автомат (ДКА, англ. Deterministic finite automaton, DFA) можно рассматривать как специальный вид НКА, в котором для любого состояния и букв алфавита функция перехода имеет лишь одно результирующее состояние. Таким образом ясно, что любой формальный язык, который можно распознать с помощью ДКА, можно распознать и с помощью НКА.

В обратную сторону для любого НКА существует ДКА, распознающий тот же самый формальный язык. ДКА может быть построен[en] с помощью конструкции подмножеств[en].

Этот результат показывает, что НКА, несмотря на его большую гибкость, не в состоянии распознать языки, которые нельзя распознать никаким ДКА. Это важно также на практике, чтобы конвертировать более простые

по структуре НКА в более эффективные в вычислительном отношении ДКА. Однако, если НКА имеет n состояний, результирующий ДКА может иметь до 2^n состояний, что иногда делает построение непрактичным для больших НКА.

Недетерминированный конечный автомат с ε-переходами (НКА-ε) является дальнейшим обобщением уже для НКА. В этом автомате функции переходов разрешается иметь пустую строку ε в качестве входа. Переход без использования входного символа называется ε-переходом. В диаграмме состояний эти переходы обычно помечаются греческой буквой ε. ε-переходы дают удобный способ моделирования систем, текущее состояние которых в точности не известно. Например, если мы моделируем систему, текущее состояние которой не ясно (после обработки некоторой входной строки) и может быть либо q, либо q', мы можем добавить ε-переход между этими двумя состояниями, переводя автомат в оба состояния одновременно.

НКА-ε формально представляется 5-кортежем, $(Q, \Sigma, \Delta, q_0, F)$, который состоит из:

конечного множества состояний Q

конечного множества входных символов, называемого алфавитом Σ

функции переходов $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to P(Q)$

начального (или стартового) состояния $q_0 \in Q$

набора состояний F, считающихся допустимыми (или конечными) состояниями $F \subseteq Q$.

Для состояния $q \in Q$ пусть E(q) означает множество состояний, достижимых из qq следующими ε-переходами в функции переходов Δ, а именно, $p \in E(q)$ если существует последовательность состояний $q_1, \ldots, q_k$ таких, что:

$q_1 = q$,

$q_{i+1} \in \Delta(q_i, \varepsilon)$ для любых $1 \leqslant i < k$

$q_k = p$.

Множество E(q) известно как ε-замыкание состояния q.

ε-замыкание определяется также для множества состояний. ε-замыкание множества состояний P НК-автомата определяется как множество состояний, достижимых из элементов множества P по ε-переходам.

Пусть $w = a_1a_2\ldots a_n$ будет строкой над алфавитом Σ. Автомат M принимает строку w, если существует последовательность состояний $r_0, r_1, \ldots, r_n$ в Q со следующими условиями:

$r_0 \in E(q_0)$

$r_{i+1} \in E(\Delta(r_i, a_{i+1}))$ для любого $i = 0, \ldots, n - 1$

$r_n \in F$.

Чтобы показать, что НКА-ε эквивалентен НКА, сначала обратим внимание на то, что НКА является частным случаем НКА-ε, остаётся показать, что для любого НКА-ε существует эквивалентный НКА.

Пусть A = (Q, Σ, Δ, q_0 , F) будет НКА-ε. НКА A’ = (Q, Σ, Δ’, E(q_0), F) эквивалентен A, где для любого a ∈ Σ и q ∈ Q верно Δ’(q, a) = E(Δ(q, a)).

Тогда НКА-ε эквивалентен НКА. Поскольку НКА эквивалентен ДКА, НКА-ε также эквивалентен ДКА

5.2. Всюду определенные конечные автоматы. Тупиковые состояния конечного автомата

Конечный автомат называют полностью определенным (всюду определенным), если в каждом его возможном состоянии функция перехода определена для всех возможных входных символов.

Теорема Пусть M = (Q, V, f, q_0, F) – детерминированный конечный автомат, не являющийся всюду определенным.

Тогда существует всюду определенный детерминированный конечный автомат M'=(Q',V, f ',q_0, F), такой что L(M) = L(M').

Доказательство. Дополним множество Q состояний ДКА новым состоянием: Q' = Q∪{q'}, q'∉Q.

Определим новую функцию f ' переходов ДКА следующим образом:

- f '(a, q) = f (a, q), если f (a, q) ≠ ∅
- f '(a, q) = {q' }, если f (a, q) = ∅
- f '(a, q') = {q' }

Легко показать, что построенный таким образом автомат допускает тот же самый язык.

Замечание. Если в какой-то момент из текущего состояния нет перехода по считанному символу, то будем считать, что автомат не допускает данное слово. При реализации вместо отдельного рассмотрения данного случая иногда удобно вводить фиктивную нетерминальную «дьявольскую вершину» (также тупиковое состояние, сток), из которой любой переход ведет в неё же саму, и заменить все несуществующие переходы на переходы в «дьявольскую вершину».

5.3. Детерминированные конечные автоматы. Способы задания детерминированных конечных автоматов

Детерминированный конечный автомат (ДКА, DFA, англ. deterministic finite automaton, DFSA, англ. deterministic finite-state automaton, DFSM англ. deterministic finite-state machine), известный также как детерминированный конечный распознаватель — это конечный автомат, принимающий или отклоняющий заданную строку символов путём прохождения через последовательность состояний, определённых строкой. Имеет единственную последовательность состояний во время работы. Мак-Каллок и Уолтер

Питтс были одними из первых исследователей, предложивших концепцию, похожую на конечный автомат в 1943 году.

Рисунок иллюстрирует детерминированный конечный автомат с помощью диаграммы состояний. В этом примере имеется три состояния – S_0, S_1 и S_2 (отражены на рисунке окружностями). Автомат на вход принимает конечную последовательность нулей и единиц. Для каждого состояния существует стрелка перехода, ведущая из состояния в другое состояние как для 0, так и для 1. После чтения символа, ДКА детерминированно переходит из одного состояния в другое, следуя по стрелке перехода. Например, если автомат находится в состоянии S_0, а входным символом является 1, то автомат детерминированно переходит в состояние S_1. ДКА имеет начальное состояние (графически показывается стрелкой «из ниоткуда»), откуда начинается вычисление, и множество конечных состояний (обозначаемых графически в виде двойной окружности), которые определяют, успешно ли закончились вычисления.

ДКА определяется как абстрактная математическая концепция, но часто реализуется в аппаратном и программном обеспечении для решения специфичных задач. Например, ДКА может моделировать программы, которые решают допустим ли введённый пользователем email адрес.

ДКА распознаёт в точности множество регулярных языков, которые, среди прочего, полезны для лексического анализа и сопоставления с образцом. ДКА могут быть построены из недетерминированного конечного автомата (НКА, англ. nondeterministic finite automata, NFAs) с помощью сведения НКА к ДКА.

Задать конечный автомат можно (с точностью до эквивалентности) описав каким либо образом функцию переходов, указав при этом начальное и заключительные состояния. Задать функцию переходов можно таблично или с помощью диаграммы состояний.

Диаграмма состояний (диаграмма переходов) – ориентированный граф для конечного автомата, в котором

вершины обозначают состояния

дуги показывают переходы между двумя состояниями

На практике вершины обычно изображаются в виде окружностей и, в случае заключительных состояний, двойных окружностей. Начальное состояние указывается с помощью стрелки, начало которой не связано ни с каким состоянием.

5.4. Эквивалентность конечных автоматов. Построение эквивалентного детерминированного конечного автомата

Как уже было показано ранее все конечные автоматы (НКА-ε, НКА, ДКА) эквивалентны. То есть, для любого из указанных автоматов существует эквивалентный, из принадлежащих к любому другому классу.

Рассмотрим алгоритм построения ДКА, эквивалентного данному НКА.

1. Пусть исходный НКА имеет k состояний. Для построения ДКА возьмем 2^k – 1состояний, каждое из которых соответствует одному элементу множества всех подмножеств состояний исходного автомата, кроме пустого множества.
2. Из каждого состояния S нового автомата направим не более чем один переход, помеченный данным символом, в такое состояние, которое соответствует множеству состояний НКА, в которые есть переходы по этому символу хотя бы из одного состояния НКА, образующего 5.
3. В качестве начального отметим состояние ДКА, имеющее то же обозначение, что и начальное состояние исходного НКА. Как конечные отметим все состояния ДКА, в которые входит хотя бы одно из конечных состояний исходного НКА.

Заметим, что не все из полученных состояний ДКА будут достижимыми. Недостижимые состояния конечного автомата можно исключить.

Есть и другой алгоритм, основанный на поочередном получении состояний ДКА как множеств состояний НКА получаемых переходами при чтении входных символов.

5.5. Минимизация детерминированного конечного автомата

Полная таблица переходов КА может быть очень большой, поэтому обычно используют различные алгоритмы минимизации.

Все алгоритмы минимизации основаны на объединении нескольких различных состояний КА в одно.

Алгоритм минимизации

Шаг 1. Разбиваем множество состояний Q исходного автомата на два класса:

множество заключительных состояний F;

множество незаключительных состояний Q \ F;

Шаг 2. Все имеющиеся классы эквивалентности разбиваем на подклассы.

Семейство всех полученных подклассов становиться новым текущим семейством классов.

Шаг 3. Если на шаге 2 разбиение изменилось, то переходим к шагу 2.

Состояния p и q исходного конечного автомата будут будут принадлежать одному и тому же подклассу, если

$\forall a\ (a \in V \Rightarrow f(p, a) \sim f(q,a))$,

где

V – алфавит исходного автомата,

f – его функция переходов,

~ – отношение эквивалентности по текущему разбиению.

6. РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

6.1. Регулярные языки и выражения. Операции над регулярными языками

Регулярный язык (регулярное множество) в теории формальных языков – множество слов, которое распознает некоторый конечный автомат. Класс регулярных множеств удобно изучать в целом, а полученные результаты оказываются применимы для достаточно широкого спектра формальных языков.

Пусть Σ – конечный алфавит. Регулярными языками в алфавите Σ называются множества слов, определяемые по индукции следующим образом:

Пустое множество ($\emptyset$) является регулярным языком.

Множество, состоящее из одной лишь пустой строки ($\{\varepsilon\}$) является регулярным языком.

Множество, состоящее из одного однобуквенного слова ($\{a\}$, где $a \in \Sigma$) является регулярным языком.

Если α и β – регулярные языки, то их объединение ($\alpha \cup \beta$), конкатенация ($\alpha\,\beta$) и замыкание Клини (α^*) тоже являются регулярными языками.

Других регулярных языков нет.

Доказано несколько теорем вида «если определенные языки регулярны, а язык L построен из них с помощью определенных операций (например, L есть объединение двух регулярных языков), то язык L также регулярен». Эти теоремы часто называют свойствами замкнутости регулярных языков, так как в них утверждается, что класс регулярных языков замкнут относительно определенных операций. Свойства замкнутости выражают идею того, что если один или несколько языков регулярны, то языки, определенным образом связанные с ним (с ними), также регулярны. Кроме того, данные свойства служат интересной иллюстрацией того, как эквивалентные представления регулярных языков (автоматы и регулярные выражения) подкрепляют друг друга в нашем понимании этого класса языков, так как часто один способ представления намного лучше других подходит для доказательства некоторого свойства замкнутости.

Основные свойства замкнутости регулярных языков выражаются в том, что эти языки замкнуты относительно следующих операций.

1. Объединение.
2. Пересечение.
3. Дополнение.
4. Разность.
5. Обращение.
6. Итерация (звездочка, замыкание).
7. Конкатенация.

8. Гомоморфизм (подстановка цепочек вместо символов языка).
9. Обратный гомоморфизм.

В основе доказательства таких свойств замкнутости лежит построение конечных автоматов.

6.2. Эквивалентные преобразования регулярных выражений

Средством записи регулярных множеств являются регулярные выражения.

Регулярное выражение в алфавите V определяется рекурсивно следующим образом:

(1) $\emptyset$ – регулярное выражение, обозначающее множество $\emptyset$;

(2) ε – регулярное выражение, обозначающее множество $\{\varepsilon\}$;

(3) a – регулярное выражение, обозначающее множество {a};

(4) если p и q – регулярные выражения, обозначающие регулярные множества P и Q соответственно, то

(а) (p|q) – регулярное выражение, обозначающее регулярное множество $P \cup Q$,

(б) (pq) – регулярное выражение, обозначающее регулярное множество PQ,

(в) (p*) – регулярное выражение, обозначающее регулярное множество P*;

(5) ничто другое не является регулярным выражением в алфавите V.

Два выражения с переменными являются эквивалентными, если при подстановке любых языков вместо переменных оба выражения представляют один и тот же язык.

Существует ряд алгебраических законов, позволяющих осуществлять эквивалентное преобразование регулярных выражений.

Пусть p, q и r – регулярные выражения. Тогда справедливы следующие соотношения:

(1) $p|q = q|p$;

(2) $\emptyset^* = \varepsilon$;

(3) $p|(q|r) = (p|q)|r$;

(4) $p(qr) = (pq)r$;

(5) $p(q|r) = pq|pr$;

(6) $(p|q)r = pr|qr$;

(7) $p\varepsilon = \varepsilon p = p$;

(8) $\emptyset p = p\emptyset = \emptyset$;

(9) $p^* = p|p^*$;

(10) $(p^*)^* = p^*$;

(11) $p|p = p$;

(12) $p|\emptyset = p$.

6.3. Лемма о накачке для регулярных языков

Лемма о накачке (лемма о разрастании, лемма-насос; англ. pumping lemma) – важное утверждение теории автоматов, позволяющее во многих случаях проверить, является ли данный язык автоматным. Поскольку все конечные языки являются автоматными, эту проверку имеет смысл делать только для бесконечных языков. Термин «накачка» в названии леммы отражает возможность многократного повторения некоторой подстроки в любой строке подходящей длины любого бесконечного автоматного языка.

Пусть L – регулярный язык над алфавитом Σ, тогда существует такое n, что для любого слова ω∈L длины не меньше n найдутся слова x,y,z∈Σ^*, для которых верно: xyz=ω,y≠ε,|xy|⩽n и ∀k⩾0 xy^kz∈L.

Если язык L является регулярным, то существует число n⩾1 такое что для любого слова uwv из языка L, где |w|⩾n может быть записано в форме uwv=uxyzv, где слова x, y и z такие, что |xy|⩽n, |y|⩾1 и uxy^izv принадлежит языку L для любого целого числа i⩾0.

Эти условия не являются достаточными для регулярности языка.

Для доказательства нерегулярности языка часто удобно использовать отрицание леммы о разрастании. Пусть L – язык над алфавитом Σ. Если для любого натурального n найдётся такое слово ω из данного языка, что его длина будет не меньше n и при любом разбиении на три слова x,y,z такие, что y непустое и длина xy не больше n, существует такое k, что xykz∉L, то язык L нерегулярный.

6.4. Построение эквивалентного регулярному выражению конечного автомата

Преобразование проводится структурной индукцией по выражению R, следуя рекурсивному определению регулярных выражений.

Построить дерево операций для регулярного выражения

Построить конечные автоматы для листьев

Построить составной конечный автомат для внутренних узлов

Для каждого символа из базового алфавита создаём лист

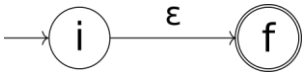
Рассматриваем операции в порядке выполнения с учетом приоритета:

- Создаём узел для рассматриваемой операции
- Делаем соответствующие операндам узлы дочерними

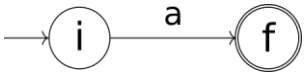
Конечные автоматы для базовых регулярных выражений

Для ∅

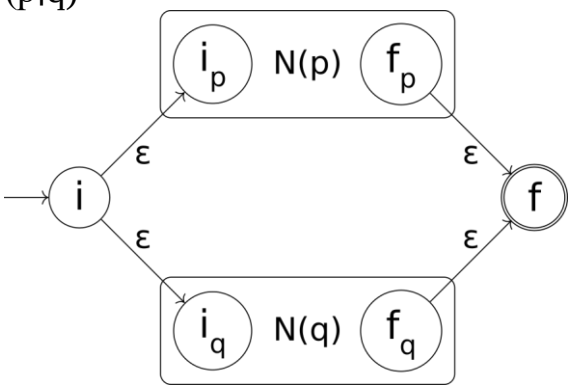
Для ε



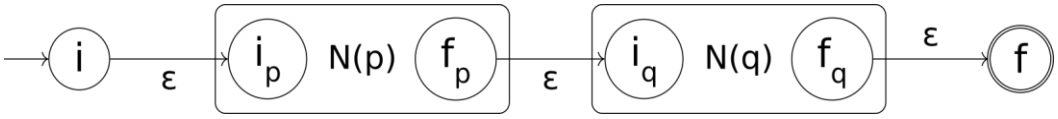
Для a



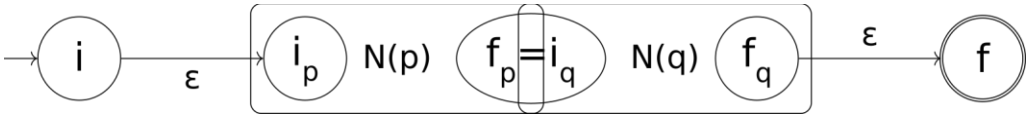
Для объединения (p|q)



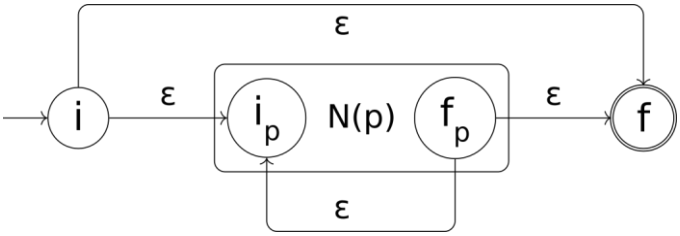
Для конкатенации (pq)



или, объединив состояния f_p и i_q



Для итерации (p*)



7. АВТОМАТЫ С МАГАЗИННОЙ ПАМЯТЬЮ

7.1. Определение автоматов с магазинной памятью

В теории автоматов, автомат с магазинной памятью – это автомат, который использует стек для хранения состояний.

Магазинный автомат – это, по существу, недетерминированный конечный автомат с ε-переходами и одним дополнением – магазином, в котором хранится цепочка «магазинных символов». Присутствие магазина означает, что в отличие от конечного автомата магазинный автомат может “помнить” бесконечное количество информации. Однако в отличие от универсального компьютера, который также способен запоминать неограниченные объемы информации, магазинный автомат имеет доступ к информации в магазине только с одного его конца в соответствии с принципом “последним пришел – первым ушел” (“last-in-first-out”).

Вследствие этого существуют языки, распознаваемые некоторой программой компьютера и нераспознаваемые ни одним магазинным автоматом. В действительности, магазинные автоматы распознают в точности КС-языки.

Автомат с магазинной памятью является набором

$M = (K, \Sigma, \pi, s, F, S, e)$,

где

K – конечное множество состояний автомата,

$s \in K$ – единственно допустимое начальное состояние автомата,

$F \subseteq K$ – множество конечных состояний, причём допустимо $F=\varnothing$ и $F=K$,

Σ – допустимый входной алфавит, из которого формируются строки, считываемые автоматом,

S – алфавит памяти (магазина),

$e \in S$ – нулевой символ памяти.

Память работает как стек, то есть для чтения доступен последний записанный в неё элемент. Таким образом, функция перехода является отображением $\pi : K \times \Sigma \times S \to K \times S$. То есть, по комбинации текущего состояния, входного символа и символа на вершине магазина автомат выбирает следующее состояние и, возможно, символ для записи в магазин. В случае, когда в правой части автоматного правила присутствует e, в магазин ничего не добавляется, а элемент с вершины стирается. Если магазин пуст, то срабатывают правила с e в левой части.

Класс языков, распознаваемых автоматами с магазинной памятью, совпадает с классом контекстно-свободных языков.

В чистом виде автоматы с магазинной памятью используются крайне редко. Обычно эта модель используется для наглядного представления

отличия обычных конечных автоматов от синтаксических грамматик. Реализация автоматов с магазинной памятью отличается от конечных автоматов тем, что текущее состояние автомата сильно зависит от любого предыдущего.

Сейчас у нас есть лишь неформальное понятие того, как МП-автомат “вычисляет”. Интуитивно МП-автомат переходит от конфигурации к конфигурации в соответствии с входными символами (или ε), но в отличие от конечного автомата, о котором известно только его состояние, конфигурация МП-автомата включает как состояние, так и содержимое магазина. Поскольку магазин может быть очень большим, он часто является наиболее важной частью конфигурации. Полезно также представлять в качестве части конфигурации непрочитанную часть входа.

Таким образом, конфигурация МП-автомата представляется тройкой (q, w, γ), где q – состояние, w – оставшаяся часть входа, γ – содержимое магазина. По соглашению вершина магазина изображается слева, а дно – справа. Такая тройка называется конфигурацией МП-автомата, или его мгновенным описанием, сокращенно МО (instantaneous description – ID).

Поскольку МО конечного автомата – это просто его состояние, для представления последовательностей конфигураций, через которые он проходил, было достаточно использовать δ . Однако для МП-автоматов нужна нотация, описывающая изменения состояния, входа и магазина. Таким образом, используются пары конфигураций, связи между которыми представляют переходы МП-автомата.

Пусть $P = (Q, \Sigma, \Gamma, \delta, q0, Z0, F)$ – МП-автомат. Определим отношение $\vdash_P$, или просто $\vdash$, когда P подразумевается, следующим образом. Предположим, что $\delta(q, a, X)$ содержит (p, α). Тогда для всех цепочек w из Σ^* и β из Γ^* полагаем $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$.

Этот переход отражает идею того, что, прочитывая на входе символ a, который может быть ε, и заменяя X на вершине магазина цепочкой α , можно перейти из состояния q в состояние p. Заметим, что оставшаяся часть входа (w) и содержимое магазина под его вершиной (β) не влияют на действие МП-автомата; они просто сохраняются, возможно, для того, чтобы влиять на события в дальнейшем.

Используем также символ $\vdash_P^*$, или просто $\vdash^*$, когда МП-автомат P подразумевается, для представления нуля или нескольких переходов МП-автомата. Итак, имеем следующее индуктивное определение.

Базис. $I \vdash^* I$ для любого МО I.

Индукция. $I \vdash^* J$, если существует некоторое МО K, удовлетворяющее условиям $I \vdash K$ и $K \vdash^* J$.

Таким образом, $I \vdash^* J$, если существует такая последовательность МО $K_1, K_2, \ldots, K_n$, у которой $I = K_1$, $J = K_n$, и $K_i \vdash K_{i+1}$ для всех $i = 1, 2, \ldots, n - 1$.

7.2. Языки автоматов с магазинной памятью

Мы предполагали, что МП-автомат допускает свой вход, прочитывая его и достигая заключительного состояния. Такой подход называется «допуск по заключительному состоянию». Существует другой способ определения языка МП-автомата, имеющий важные приложения. Для любого МП-автомата мы можем определить язык, «допускаемый по пустому магазину», т.е. множество цепочек, приводящих МП-автомат в начальной конфигурации к опустошению магазина.

Эти два метода эквивалентны в том смысле, что для языка L найдется МП-автомат, допускающий его по заключительному состоянию тогда и только тогда, когда для L найдется МП-автомат, допускающий его по пустому магазину. Однако для конкретных МП-автоматов языки, допускаемые по заключительному состоянию и по пустому магазину, обычно различны. В этом разделе показывается, как преобразовать МП-автомат, допускающий L по заключительному состоянию, в другой МП-автомат, который допускает L по пустому магазину, и наоборот.

Пусть $P = (Q, \Sigma, \Gamma, \delta, q0, Z0, F)$ – МП-автомат. Тогда L(P), языком, допускаемым P по заключительному состоянию, является

$$\{w \mid (q0, w, Z0) \vdash_P^* (q, \varepsilon, \alpha)\}$$

для некоторого состояния q из F и произвольной магазинной цепочки α. Таким образом, начиная со стартовой конфигурации с w на входе, P прочитывает w и достигает допускающего состояния. Содержимое магазина в этот момент не имеет значения.

Для каждого МП-автомата $P = (Q, \Sigma, \Gamma, \delta, q0, Z0, F)$ определим

$$N(P) = \{w \mid (q0, w, Z0) \vdash_P^* (q, \varepsilon, \varepsilon)\},$$

где q – произвольное состояние. Таким образом, N(P) представляет собой множество входов w, которые P может прочитать, одновременно опустошив свой магазин.

Поскольку множество допускающих состояний не имеет значения, иногда в случаях, когда нас будет интересовать допуск только по пустому магазину, будем записывать МП-автомат в виде шестерки $(Q, \Sigma, \Gamma, \delta, q0, Z0)$, опуская седьмой компонент.

7.3. Соотношение между регулярными языками, КС-языками и языками детерминированных МП-автоматов

Детерминированным автоматом с магазинной памятью (англ. deterministic pushdown automaton) называется автомат с магазинной памятью, для которого выполнены следующие условия:

$q \in Q, a \in \Sigma \cup \{\varepsilon\}, X \in \Gamma \Rightarrow \delta(q,a,X)$ имеет не более одного элемента – $\delta: Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \to Q \times \Gamma^*$.

Если $\delta(q,a,X)$ не пусто для некоторого $a \in \Sigma$, то $\delta(q,\varepsilon,X)$ должно быть пустым.

В отличие от конечных автоматов, для МП-автоматов недетерминизм является существенным. ДМП-автоматы распознают не все языки, распознаваемые МП-автоматами или КС-грамматиками.

ДМП-автоматы допускают класс языков, который находится между регулярными и КС-языками. Вначале докажем, что языки ДМП-автоматов включают в себя все регулярные.

Теорема Если L – регулярный язык, то L = L(P) для некоторого ДМП-автомата P.

Если мы хотим, чтобы ДМП-автомат допускал по пустому магазину, то обнаруживаем, что возможности по распознаванию языков существенно ограничены. Говорят, что язык L имеет префиксное свойство, или свойство префиксности, если в L нет двух различных цепочек x и y, где x является префиксом y.

7.4. Лемма о накачке для контекстно-свободных языков

В этом разделе развивается инструмент доказательства, что некоторые языки не являются контекстно-свободными. Теорема, называемая “леммой о накачке для контекстно-свободных языков”1, гласит, что в любой достаточно длинной цепочке КС-языка можно найти две близлежащие короткие подцепочки (одна из них может быть пустой) и совместно их “накачивать”. Таким образом, обе подцепочки можно повторить i раз для любого целого i, и полученная цепочка также будет принадлежать языку.

Эта теорема отличается от аналогичной для регулярных языков, гласящей, что всегда можно найти одну короткую цепочку для ее накачки. Разница видна, если рассмотреть язык типа $L = \{0^n1^n \mid n \geq 1\}$. Можно показать, что он нерегулярен, если зафиксировать n и накачать подцепочку из нулей, получив цепочку, в которой символов 0 больше, чем 1. Однако лемма о накачке для КС-языков утверждает, что можно найти две короткие цепочки, поэтому нам пришлось бы использовать для накачки цепочку из нулей и цепочку из единиц, порождая таким образом только цепочки из L. Этот результат нас устраивает, так как L – КС-язык, а для построения цепочек, не принадлежащих языку L, лемма о накачке для КС-языков использоваться и не должна.

Первый шаг на пути к лемме о накачке для КС-языков состоит в том, чтобы рассмотреть вид и размер деревьев разбора. Одно из применений НФХ – преобразовывать деревья разбора в бинарные (двоичные). Такие деревья имеют ряд удобных свойств, и одно из них используется здесь.

Пусть дано дерево разбора, соответствующее НФХ-грамматике (грамматике в нормальной форме Хомского) G = (V, T, P, S), и пусть кроной дерева является терминальная цепочка w. Если n – наибольшая длина пути (от корня к листьям), то $|w| \leq 2^{n-1}$.

Пусть L – контекстно-свободный язык над алфавитом Σ, тогда существует такое n, что для любого слова $\omega \in L$ длины не меньше n найдутся слова $u,v,x,y,z \in \Sigma*$, для которых верно: $uvxyz=\omega, vy \neq \varepsilon, |vxy| \leqslant n$ и $\forall k \geqslant 0$ $uv^kxy^kz \in L$.

7.5. Свойства замкнутости и разрешимости контекстно-свободных языков

Рассмотрим некоторые операции над контекстно-свободными языками, которые гарантированно порождают КС-языки. Многие из этих свойств замкнутости соответствуют теоремам для регулярных языков. Однако есть и отличия.

Вначале введем операцию подстановки, по которой каждый символ в цепочках из одного языка заменяется целым языком. Эта операция обобщает гомоморфизм, рассмотренный в разделе 4.2.3, и является полезной в доказательстве свойств замкнутости КС-языков, относительно некоторых других операций, например, регулярных (объединение, конкатенация и замыкание). Покажем, что КС-языки замкнуты относительно гомоморфизма и обратного гомоморфизма. В отличие от регулярных языков, КС-языки не замкнуты относительно пересечения и разности. Однако пересечение или разность КС-языка и регулярного языка всегда является КС-языком.

Пусть Σ – алфавит. Предположим, для каждого символа a из Σ выбран язык L_a. Выбранные языки могут быть в любых алфавитах, не обязательно Σ и не обязательно одинаковых. Выбор языков определяет функцию s (подстановка, substitution) на Σ, и L_a обозначается как s(a) для каждого символа a.

Если $w = a_1a_2\ldots a_n$ – цепочка из Σ^*, то s(w) представляет собой язык всех цепочек $x_1x_2\ldots x_n$, у которых для i = 1, 2, …, n цепочка x_i принадлежит языку $s(a_i)$. Иными словами, s(w) является конкатенацией языков $s(a_1)s(a_2)\ldots s(a_n)$. Определение s можно распространить на языки: s(L) – это объединение s(w) по всем цепочкам w из L.

Если L – КС-язык в алфавите Σ, а s – подстановка на Σ, при которой s(a) является КС-языком для каждого a из Σ, то s(L) также является КС-языком.

Контекстно-свободные языки замкнуты относительно следующих операций.

1. Объединение. 2. Конкатенация. 3. Замыкание (*) и транзитивное замыкание (+). 4. Гомоморфизм.

КС-языки замкнуты также относительно обращения.

КС-языки не замкнуты по пересечению.

Если L – КС-язык, а R – регулярный язык, то $L \cap R$ является КС-языком.

Разрешимых вопросов, связанных с КС-языками, совсем немного. Основное, что можно сделать, – это проверить, пуст ли язык, и принадлежит ли данная цепочка языку.

Список наиболее значительных неразрешимых вопросов о контекстно-свободных грамматиках и языках.

1. Неоднозначна ли данная КС-грамматика G?
2. Является ли данный КС-язык существенно неоднозначным (то есть, таким, для которого не существует однозначной грамматики)?
3. Пусто ли пересечение двух КС-языков?
4. Равны ли два данных КС-языка?
5. Равен ли Σ* данный КС-язык, где Σ – алфавит этого языка?

8. СИНТАКСИЧЕСКИЙ АНАЛИЗ

8.1. Назначение и принципы работы синтаксического анализатора. Распознаватели

В нашей модели компилятора синтаксический анализатор получает строку токенов от лексического анализатора и проверяет, может ли эта строка имен токенов порождаться грамматикой исходного языка. Мы также ожидаем от синтаксического анализатора сообщений обо всех выявленных ошибках, причем достаточно внятных и полных, а кроме того, умения обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы. Концептуально в случае корректной программы синтаксический анализатор строит дерево разбора и передает его следующей части компилятора для дальнейшей обработки. В действительности явное построение дерева разбора не требуется, поскольку проверки и действия трансляции, как мы уже видели,могут выполняться в процессе синтаксического анализа. Таким образом,синтаксический анализатор и прочие части начальной стадии компилятора могут быть реализованы в виде единого модуля.

Имеется три основных типа синтаксических анализаторов грамматик:универсальные, восходящие и нисходящие. Универсальные методы разбора, такие как алгоритмы Кока-Янгера-Касами (Cocke-Younger-Kasami) и Эрли (Earley), могут работать с любой грамматикой. Однако эти обобщенные методы слишком неэффективны для использования в промышленных компиляторах.

Методы, обычно применяемые в компиляторах, можно классифицировать как нисходящие (сверху вниз – top-down) или восходящие (снизу вверх – bottom-up). Как явствует из названий, нисходящие синтаксические анализаторы строят дерево разбора сверху (от корня) вниз (к листьям), тогда как восходящие начинают с листьев и идут к корню. В обоих случаях входной поток синтаксического анализатора сканируется посимвольно слева направо.

Наиболее эффективные нисходящие и восходящие методы работают только с подклассами грамматик, однако некоторые из этих классов, такие как LL- и LR-грамматики, достаточно выразительны для описания большинства синтаксических конструкций языков программирования. Реализованные вручную синтаксические анализаторы чаще работают с LL-грамматиками; например, предиктивный подход работает с LL-грамматиками. Синтаксические анализаторы для большего класса LR-грамматик обычно создаются с помощью автоматизированных инструментов.

Будем полагать, что выход синтаксического анализатора является некоторым представлением дерева разбора потока токенов от лексического анализатора. На практике имеется множество задач,которые могут сопровождать процесс разбора, такие как сбор информации о различных токенах

в таблицу символов, выполнение проверки типов и других видов семантического анализа,а также генерация промежуточного кода.

Для произвольной КС-грамматики может быть построен недетерминированный автомат с магазинной памятью, принимающий язык, порождаемый этой грамматикой.

После того как ошибка обнаружена, каким образом должно выполниться восстановление синтаксического анализатора? Хотя универсальной стратегии не существует, все же несколько методов применяются чаще других. Простейший подход состоит в том, что синтаксический анализатор завершает работу при первой же обнаруженной ошибке, выводя о ней информативное сообщение. Если же синтаксический анализатор может восстановить некоторым образом свое состояние до такого, когда работа может быть продолжена, то имеется определенная надежда на то, что будут обнаружены и другие ошибки, а также что информация о них окажется достаточно корректной. Если же восстановление не совсем корректное и количество обнаруженных ошибок растет, как снежный ком, то будет лучше,если компилятор после некоторого предельного количества ошибок прекратит вывод раздражающих сообщений об этих «фальшивых» ошибках.

Восстановление в режиме паники. В этом случае при обнаружении ошибки синтаксический анализатор пропускает входные символы по одному, пока не будет найден один из специально определенного множества синхронизирующих токенов.

Восстановление на уровне фразы. При обнаружении ошибки синтаксический анализатор может выполнить локальную коррекцию оставшегося входного потока, т.е. он может заменить префикс остальной части потока некоторой строкой, которая позволит синтаксическому анализатору продолжить работу.

Знание наиболее распространенных ошибок позволяет расширить грамматику языка продукциями, порождающими ошибочные конструкции. Синтаксический анализатор, построенный на основе такой расширенной продукциями ошибок грамматики языка, обнаруживает ошибку при использовании "ошибочной"продукции. Таким образом, он может сгенерировать корректное диагностическое сообщение для распознанной во входном потоке ошибки.

В идеальном случае при обработке некорректной входной строки компилятор должен вносить минимально возможное количество изменений. Существует ряд алгоритмов, которые позволяют выбрать минимальную последовательность вносимых изменений для проведения коррекции. По заданной некорректной строке x и грамматике G такие алгоритмы находят дерево разбора для строки y, такой,что количество вставок, удалений и изменений токенов, требуемых дляпреобразования x в y, минимально возможное. К сожалению эти методы в общем случаеслишком дорогостоящи для применения в компиляторах в силу высоких требо-ваний к памяти и времени работы, а потому представляют в настоящее время,в основном, теоретический интерес.

Следует заметить, что ближайшая к исходной исправленная программа может оказаться вовсе не тем, что хотел получить программист. Тем не менее понятие минимальной коррекции дает нам критерий оценки качества различных технологий восстановления после ошибок и используется для поиска оптимальной замены строки при восстановлении на уровне фразы.

8.2. Нисходящий синтаксический анализ. Метод рекурсивного спуска

Нисходящий синтаксический анализ можно рассматривать как задачу построения дерева разбора для входной строки, начиная с корня и создавая узлы дерева разбора в прямом порядке обхода. Или, что то же самое, нисходящий синтаксический анализ можно рассматривать как поиск левого порождения входной строки.

На каждом шаге нисходящего синтаксического анализа ключевой проблемой является определение продукции, применимой для нетерминала, скажем, A. Когда A-продукция выбрана, остальная часть процесса синтаксического анализа состоит из проверки "соответствий" терминальных символов в теле продукции входной строке.

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

Рекурсивный спуск в общем случае может потребовать выполнения возврата,т.е. повторения сканирования входного потока. Однако при анализе синтаксических конструкций языков программирования возврат требуется редко, так что встреча с синтаксическим анализатором с возвратом – явление не частое. Даже в ситуациях наподобие синтаксического анализа естественного языка возврат не слишком эффективен, и предпочтительными являются табличные методы наподобие динамического программирования или метода Эрли (Earley).

8.3. Множества FIRST и FOLLOW

При построении как нисходящего, так и восходящего синтаксического анализатора нам помогут две функции – FIRST и FOLLOW, – связанные с грамматикой G. В процессе нисходящего синтаксического анализа FIRST и FOLLOW позволяют выбрать применяемую продукцию на основании очередного символа входного потока. Множества токенов, порождаемые функцией FOLLOW, могут также использоваться как синхронизирующие токены в процессе восстановления после ошибки в "режиме паники". Определим

FIRST (a), где а – произвольная строка символов грамматики, как множество терминалов, с которых начинаются строки, порождаемые α. Если $\alpha \Rightarrow^* \varepsilon$, то $\varepsilon \in$ FIRST (α).

Пусть с – символ из алфавита Σ, α, β – строки из нетерминалов и терминалов (возможно пустые), S, A – нетерминалы грамматики (начальный и произвольный соответственно), $ – символ окончания слова. Тогда определим FIRST и FOLLOW следующим образом:

FIRST(A)={c|A⇒∗cβ}∪{ε if A⇒*ε}

FOLLOW(A)={c|S⇒∗αAcβ}∪{$ if S⇒*αA}

Другими словами, FIRST(A) – все символы (терминалы), с которых могут начинаться всевозможные выводы из α, а FOLLOW(A) – всевозможные символы, которые встречаются после нетерминала А во всех небесполезных правилах грамматики.

Чтобы вычислить FIRST (X) для всех символов грамматики X, буде применять следующие правила до тех пор, пока ни к одному из множеств FIRST не смогут быть добавлены ни терминалы, ни б.1. Если X – терминал, то FIRST (X) = {X}.2. Если X – нетерминал и имеется продукция X –► У^Уъ ... У& для некоторого к ^ 1, то поместим а в FIRST (X), если для некоторого г а EFIRST (Yi)и б входит во все множества FIRST (Yi), ..., FIRST (Y^-i), т.е. Y\ • • •У5_1 4> б. Если б входит в FIRST (Yj) для всех j = 1,2,..., fc, то добавляем б к FIRST (X). Например, все, что находится в множестве FIRST (Yi), есть и в множестве FIRST (X). Если Y\ не порождает б, то больше мы ничего не добавляем к FIRST (X), но если Y\ 4> б, то к FIRST (X) добавляется FIRST (У2) и т.д.3. Если имеется продукция X –> б, добавим е к FIRST (X). Теперь можно вычислить FIRST для любой строки X1X2 ... Хп следующим образом. Добавим к FIRST (X1X2... Хп) все не-б-символы из FIRST (Xi).Добавим также все не-б-символы из FIRST (X2), если е GFIRST (Xi)9 все не-б-символы из FIRST (У3), если б имеется как в FIRST (Xi), так и в FIRST (X2), и т.д. И наконец, добавим б к FIRST (X1X2 ... Хп), если для всех г FIRST (Xi) содержит б. Чтобы вычислить FOLLOW (A) для всех нетерминалов А, будем применять следующие правила до тех пор, пока ни к одному множеству FOLLOW нельзя будет добавить ни одного символа.1. Поместим $ в FOLLOW (5), где S –ограничитель входного потока. стартовый символ, а $

2. Если имеется продукция А –> а В /?, то все элементы множества FIRST (/3),кроме е, помещаются в множество FOLLOW (В).3. Если имеется продукция А ^> аВ или А –* а В /?, где FIRST (/?) содержит б, то все элементы из множества FOLLOW (А) помещаются в множество FOLLOW (В).

8.4. LL-грамматики. Предиктивный нисходящий синтаксический анализатор

Предиктивные синтаксические анализаторы, т.е. синтаксические анализаторы, работающие методом рекурсивного спуска без возврата, могут быть построены для класса грамматик, называющегося LL(1). Первое "L" в LL(1) означает сканирование входного потока слева направо, второе "L" – получение левого порождения, а "1" – использование на каждом шаге предпросмотра одного символа для принятия решения о действиях синтаксического анализатора.

Класс грамматик LL(1) достаточно богат для того, чтобы охватить большинство программных конструкций, хотя при написании грамматики для исходного языка требуется аккуратность. Например, в LL(1)-грамматике не может быть ни левой рекурсии, ни неоднозначности.

Грамматика G принадлежит классу LL(1) тогда и только тогда, когда для любых двух различных продукций G $A \rightarrow \alpha \mid \beta$ выполняются следующие условия.

1. Не существует такого терминала a, для которого и α, и β порождают строку, начинающуюся с a.
2. Пустую строку может порождать не более чем одна из продукций α или β.
3. Если $\beta \Rightarrow^* \varepsilon$, то α не порождает ни одну строку, начинающуюся с терминала из FOLLOW (A). Аналогично, если $\alpha \Rightarrow^* \varepsilon$, то β не порождает ни одну строку, начинающуюся с терминала из FOLLOW(A).

Первые два условия эквивалентны утверждению, что FIRST (α) и FIRST(β) представляют собой непересекающиеся множества. Третье условие эквивалентно утверждению, что если $\varepsilon \in$ FIRST (β), то FIRST (α) и FOLLOW (A) –непересекающиеся множества; аналогичное утверждение справедливо и в случае, если $\varepsilon \in$ FIRST (α).

Для LL(l)-грамматик могут быть построены предиктивные синтаксические анализаторы, поскольку корректная продукция для применения к нетерминалу может быть выбрана путем просмотра только текущего входного символа. Конструкции управления потоком с их определяющими ключевыми словами обычно удовлетворяют ограничениям LL(1).

Нерекурсивный предиктивный синтаксический анализатор можно построить с помощью явного использования стека (вместо неявного при рекурсивных вызовах). Синтаксический анализатор имитирует левое порождение. Если w – входная строка, соответствие которой проверено до текущего момента, то в стеке хранится последовательность грамматических символов α, такая, что

$$S \Rightarrow_{lm}^* w\alpha$$

8.5. Восходящий синтаксический анализ. LR-автомат

LR-анализатор (англ. LR parser) – синтаксический анализатор для исходных кодов программ, написанных на некотором языке программирования, который читает входной поток слева (Left) направо и производит наиболее правую (Right) продукцию контекстно-свободной грамматики. Используется также термин LR(k)-анализатор, где k выражает количество непрочитанных символов предпросмотра во входном потоке, на основании которых принимаются решения при анализе. Обычно k равно 1 и часто опускается.

Синтаксис многих языков программирования может быть определён грамматикой, которая является LR(1) или близкой к этому, и по этой причине LR-анализаторы часто используются компиляторами для выполнения синтаксического анализа исходных кодов.

Обычно на анализатор ссылаются в связи с именем того языка, исходный код которого он разбирает, например, «C++ анализатор» разбирает исходные коды языка C++.

LR-анализатор может быть создан из контекстно-свободной грамматики программой, называемой генератор синтаксических анализаторов, или же написан вручную программистом. Контекстно-свободная грамматика классифицируется как LR(k), если существует LR(k)-анализатор для неё, как определено генератором анализаторов.

Говорится, что LR-анализатор выполняет разбор снизу вверх, потому что он пытается вывести продукцию верхнего уровня грамматики, строя её из листьев.

Детерминированный контекстно-свободный язык – это язык, для которого существует какая-либо LR(k) грамматика. Каждая LR(k) грамматика может быть автоматически преобразована в грамматику LR(1) для того же языка, в то время как LR(0) грамматики для некоторых языков может не существовать. LR(0)-языки являются собственным подмножеством детерминированных.

LR-анализатор основан на алгоритме, приводимом в действие таблицей анализа, структурой данных, которая содержит синтаксис анализируемого языка. Таким образом, термин LR-анализатор на самом деле относится к классу анализаторов, которые могут разобрать почти любой язык программирования, для которого предоставлена таблица анализа. Таблица анализа создаётся генератором синтаксических анализаторов.

LR-анализ может быть обобщён как произвольный анализ контекстно-свободного языка без потери производительности, даже для LR(k) грамматик. Это происходит благодаря тому, что большинство языков программирования могут быть выражены LR(k) грамматикой, где k – малая константа (обычно 1). Заметьте, что разбор не-LR(k) грамматик на порядок медленнее (кубический вместо квадратичного в отношении размера входного потока).

LR-анализ может применяться к большему количеству языков, чем LL-анализ, а также лучше в части сообщения об ошибках, то есть он определяет

синтаксические ошибки там, где вход не соответствует грамматике, как можно раньше. В отличие от этого, LL(k) (или, что хуже, даже LL(*)) анализаторы могут задерживать определение ошибки до другой ветки грамматики из-за отката, часто затрудняя определение места ошибки в местах общих длинных префиксов.

LR-анализаторы сложно создавать вручную и обычно они создаются генератором синтаксических анализаторов или компилятором компиляторов. В зависимости от того, как была создана таблица анализа, эти анализаторы могут быть названы простыми LR-анализаторами (SLR), LR-анализаторами с предпросмотром (LALR) или каноническими LR-анализаторами. LALR-анализатор имеют значительно большую распознавательную способность, чем SLR-анализаторы. При этом таблицы для SLR-анализа имеют такой же объём, что и для LALR-анализа, поэтому SLR-анализ уже не используется.

Восходящий синтаксический анализ соответствует построению дерева разбора для входной строки, начиная с листьев (снизу) и идя по направлению к корню(вверх). Удобно описывать синтаксический анализ как процесс построения дерева разбора, хотя начальная стадия компиляции может в действительности быть выполнена и без явного построения дерева.

Можно рассматривать восходящий синтаксический анализ как процесс "свертки" строки w к стартовому символу грамматики. На каждом шаге свертки (reduction) определенная подстрока, соответствующая телу продукции, заменяется нетерминалом из заголовка этой продукции.

По определению свертка представляет собой шаг, обратный порождению (вспомните, что в порождении нетерминал в сентенциальной форме замещается телом одной из его продукций). Цель восходящего синтаксического анализа, таким образом, состоит в построении порождения в обратном порядке.

Восходящий синтаксический анализ в процессе сканирования входного потока слева направо строит правое порождение в обратном порядке. Неформально говоря, основа, или дескриптор (handle), строки – это подстрока, которая соответствует телу продукции и свертка которой представляет собой один шаг правого порождения в обратном порядке.

Формально, если $S \Rightarrow_{rm}^{*} \alpha A w \Rightarrow_{rm} \alpha\beta w$, то продукция $A \rightarrow \beta$ в позиции после α является основой (handle) $\alpha\beta w$.

Заметим, что строка w справа от основы должна содержать только терминальные символы. Для удобства мы будем говорить как об основе о теле продукции β, а не обо всей продукции $A \rightarrow \beta$ в целом. Следует также заметить, что грамматика может быть неоднозначной, с несколькими правыми порождениями $\alpha\beta w$. Если грамматика однозначна, то каждая правосентенциальная форма грамматики имеет ровно одну основу.

Обращенное правое порождение может быть получено посредством "обрезки основ". Мы начинаем процесс со строки терминалов^, которую хотим проанализировать. Если w – предложение рассматриваемой грамматики, то пусть $w = \gamma_n$ где γ_n – n-я правосентенциальная форма некоторого еще неизвестного правого порождения $S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$.

Для воссоздания этого порождения в обратном порядке мы находим основу β_n в γ_n и заменяем ее левой частью продукции $A_n \to \beta_n$ для получения предыдущей правосентенциальной формы γ_{n-1}.

Затем мы повторяем описанный процесс. Если после очередного шага право сентенциальная форма содержит только стартовый символ, мы прекращаем процесс и сообщаем об успешном завершении анализа. Обращенная последовательность продукций, использованных в свертках, представляет собой правое порождение входной строки.

LR(0)-пункт – порождающее правило с точкой в некоторой позиции правой части. Правило $A \to \varepsilon$ генерирует единственный пункт $A \to \cdot$.

Пример Порождающее правило $A \to XY\ Z$ даёт четыре пункта:

$A \to \cdot XY\ Z$

$A \to X \cdot Y\ Z$

$A \to XY \cdot Z$

$A \to XY\ Z \cdot$

Канонический набор множеств пунктов обеспечивает основу для построения автомата, используемого при принятии решений. Множества из канонического набора являются состояниями такого автомата.

При построении канонического набора используется расширенная грамматика G′, получаемая из исходной грамматики G добавлением нового стартового нетерминального символа S′ и порождающего правила $S' \to S$. Новое порождающее правило служит для определения момента завершения анализа.

Пусть I – множество пунктов грамматики G, тогда замыкание CLOSURE(I) строится по следующим правилам:

$I \subseteq CLOSURE(I)$

Если $[A \to \alpha \cdot B\beta] \in COLSURE(I)$, а $B \to \gamma$ является порождающим правилом, то $[B \to \cdot\gamma] \in CLOSURE(I)$.

Для определения переходов в LR(0)-автомате используется функция GOTO(I,X), определяемая следующим образом:

$GOTO(I, X) = CLOSURE(\{[A \to \alpha X \cdot \beta] \mid [A \to \alpha \cdot X\beta]\})$,

где I – множество пунктов, X – символ полного алфавита грамматики.

Обозначим через C канонический набор. Тогда его можно определить с помощью следующих правил:

$COLOSURE(\{[S' \to \cdot S]\}) \in C$

$(\forall I \in C)(\forall X \in V)(GOTO(I,X) \neq \emptyset \Rightarrow GOTO(I,X) \in C)$

Где V – полный алфавит грамматики.

Состояниями автомата являются элементы канонического набора множеств пунктов. Переходы задаются с помощью функции GOTO. Начальное состояние порождается пунктом $S' \to \cdot S$. Автомат допускает цепочку терминалов, если после её прочтения оказывается в состоянии, содержащем пункт $S' \to S \cdot$. Автомат не является конечным, так как обладает стеком состояний.

Заметим, что все пункты вида $B \to \cdot\gamma$ для какого-либо нетерминала B, отличного от S′, добавляются одновременно. Кроме того, замыкание

дополняется только такими элементами. Следовательно, множество используемых пунктов можно разделить на два класса:

Базисные пункты, или пункты ядра (kernel items): начальный пункт $S' \rightarrow \cdot S$ и все пункты, у которых точки расположены не у левого края.

Небазисные (nonkernel) пункты, у которых точки расположены слева, за исключением $S' \rightarrow \cdot S$.

Небазисные пункты некоторого множества из канонического набора получаются замыканием подмножества базисных пунктов этого множества.

Если для текущего состояния I и очередного входного символа X верно соотношение $GOTO(I,X) \neq \emptyset$, то выполняется сдвиг и переход в новое состояние. Если такого перехода нет, то выполняется свертка по правилу, для которого в текущем состоянии есть пункт вида $A \rightarrow \gamma\cdot$. При этом считается считанным символ A.

8.6. Разбирающие выражения грамматики

Грамматика, разбирающая выражение (РВ-грамматика) – тип аналитической формальной грамматики, описывающей формальный язык в терминах набора правил для распознавания строк языка. Грамматика, разбирающая выражение, в сущности, представляет собой синтаксический анализатор рекурсивного спуска в чисто схематической форме, которая выражает только синтаксис и не зависит от конкретной реализации или применения синтаксического анализатора. Грамматики, разбирающие выражение, похожи на регулярные выражения и на контекстно-свободные грамматики (КС-грамматики) в нотации Бэкуса-Наура, но имеют отличную от них интерпретацию.

В отличие от КС-грамматик, РВ-грамматики не могут быть неоднозначными: если строка разбирается, то существует ровно одно дерево разбора. Это делает РВ-грамматики пригодными для компьютерных языков, но не для естественных.

Формально, грамматика, разбирающая выражение, состоит из:

конечного множества N нетерминальных символов;

конечного множества Σ терминальных символов, не пересекающегося с N;

конечного множества P правил вывода;

выражения eS, называемое начальным выражением.

Каждое правило вывода из P имеет вид $A \leftarrow e$, где A – нетерминальный символ, а e – выражение разбора. Выражение разбора – это иерархическое выражение, похожее на регулярное выражение, которое строится следующим образом:

Атомарное выражение разбора состоит из: любого терминального символа; любого нетерминального символа, или пустой строки ε.

Для данных выражений разбора e, e_1 и e_2, следующие операторы порождают новые выражения разбора:

Последовательность: e_1 e_2

Упорядоченный выбор: e_1 / e_2

Нуль или более: e*

Один или более: e+

Необязательно: e?

И-предикат: &e

НЕ-предикат: !e

Фундаментальное отличие PB-грамматики от КС-грамматики заключается в том, что оператор выбора PB-грамматики является упорядоченным. Если первая альтернатива срабатывает, то все последующие – игнорируются. Таким образом, упорядоченный выбор некоммутативен, в отличие от книжных определений контекстно-свободных грамматик и регулярных выражений. Упорядоченный выбор аналогичен мягкому оператору отсечения в некоторых логических языках программирования.

Вследствие этого, при преобразовании КС-грамматики напрямую в PB-грамматику всякая неоднозначность устраняется детерминированным образом в пользу одного из возможных деревьев разбора. Аккуратно выбирая порядок указания грамматических альтернатив, программист может получить значительный контроль над выбором нужного дерева разбора.

Как и булевы контекстно-свободные грамматики, PB-грамматики имеют предикаты И- и НЕ-. Они помогают и далее устранять неоднозначность, если переупорядочивание альтернатив не может задать желаемое дерево разбора.

Каждый нетерминал в PB-грамматике, по существу, представляет собой разбирающую функцию в анализаторе рекурсивным спуском, а соответствующее выражение разбора представляет собой «код» этой функции. Каждая разбирающая функция принимает на вход строку и выдаёт один из следующих результатов:

успех, в случае которого функция может опционально передвинуть вперёд или «поглотить» один или несколько символов входной строки

провал, в случае которого вход не поглощается.

Нетерминал может завершиться успешно без поглощения ввода, и это состояние отлично от провала.

Атомарное выражение разбора, состоящее из единственного терминала, завершается успешно, если первый символ входной строки с ним совпадает, и поглощает его. Иначе результат неуспешен. Атомарное выражение из пустой строки всегда завершается успешно без поглощения. Атомарное выражение, состоящее из нетерминала A, представляет собой рекурсивный вызов функции-нетерминала A.

Оператор последовательности e_1 e_2 сначала вызывает e_1 и, если e_1 выполняется успешно, далее вызывает e_2 от части строки, оставшейся

непоглощённой e_1 и возвращает результат. Если e_1 или e_2 проваливается, то проваливается и оператор последовательности e_1 e_2.

Оператор выбора e_1 / e_2 сначала вызывает e_1 и, если e_1 успешно, возвращает её результат. Иначе, если e_1 проваливается, оператор выбора восстанавливает входную строку в состояние, предшествующее вызову e_1, и вызывает e_2, возвращая её результат.

Операторы нуль-или-более, один-или-более и необязательности поглощают соответственно нуль или более, одно или более, или нуль либо одно последовательное появление своего подвыражения e. В отличие от КС-грамматик и регулярных выражений, эти операторы всегда являются жадными, и поглощают столько входных экземпляров, сколько могут. (Регулярные выражения сначала действуют жадно, но затем в случае провала возвращаются в исходное состояние и пытаются найти более короткую последовательность). Например, выражение a* всегда поглотит все доступные символы a, а выражение (a* a) всегда провалится, поскольку после выполнения первой части a* не останется символов a для второй.

Наконец, И-предикат и НЕ-предикат реализуют синтаксические предикаты. Выражение &e вызывает подвыражение e, и возвращает успех, если e успешно, и провал в противном случае, но никогда не поглощает ввода. Аналогично, выражение !e срабатывает успешно, если e проваливается и проваливается, если e успешно, так же не поглощая ввода. Поскольку выражение e может представлять собой сколь угодно сложную конструкцию, вычисляемую «наперёд» без поглощения входной строки, эти предикаты предоставляют мощные синтаксические средства предварительного анализа и устранения неоднозначности.

Любая PB-грамматика может напрямую быть преобразована в анализатор рекурсивным спуском. Из-за неограниченной способности к предварительному анализу результирующий парсер может работать, в худшем случае, экспоненциальное время.

Запоминая результат промежуточных шагов анализа и удостоверяясь в том, что каждая разбирающая функция вызывается не более одного раза для данной позиции входных данных, можно преобразовать любую PB-грамматику в packrat-парсер, который всегда работает линейное время за счёт существенного увеличения затрат памяти.

Packrat-парсер – разновидность анализатора, работающего схожим с рекурсивным спуском методом, за исключением того, что при анализе он запоминает промежуточные результаты всех вызовов взаимно рекурсивных функций анализа. Из-за этого packrat-парсер способен анализировать множество контекстно-свободных грамматик и любую PB-грамматику (включая некоторые, порождающие не контекстно-свободные языки) в линейное время.

Также возможно построить LL-анализатор и LR-анализатор для PB-грамматик, но способность к неограниченному предварительному анализу в этом случае теряется.

9. СИНТАКСИЧЕСКИ УПРАВЛЯЕМАЯ ТРАНСЛЯЦИЯ. ПРЕОБРАЗОВАТЕЛИ

Часто, осуществляя разбор, мы хотим извлечь какие-то данные или произвести какие-то действия, а не просто выяснить, разбирается ли текст в данной грамматике. Вообще говоря, сначала можно получить дерево разбора, а потом уже, обходя его, выполнять эти действия. В этом случае происходит дублирование функционала: промежуточное сохранение данных в виде дерева разбора не нужно, а иногда его просто слишком расточительно хранить в памяти целиком. В связи с этим хочется какие-то действия производить уже на этапе разбора.

Например, мы хотим не только построить дерево разбора для арифметических выражений, а ещё и вычислить значение этого выражения. Возможно, даже не строя само дерево разбора.

Такой подход называется синтаксически управляемой трансляцией.

9.1. Синтаксически управляемые определения. Атрибутные грамматики. Аннотированные деревья разбора

Синтаксически управляемое определение (англ. syntax-directed definition) является контекстно-свободной грамматикой с атрибутами и правилами. Атрибуты связаны с грамматическими символами, а правила – с продукциями.

Синтаксически управляемая трансляция (англ. syntax-directed translation) – это трансляция, при которой в процессе разбора строки сразу выполняются какие-то действия, без использования промежуточного представления в виде дерева разбора.

Синтаксически управляемая трансляция вводит две новые сущности: атрибут и транслирующий символ.

Атрибут (англ. attribute) – дополнительные данные, ассоциированные с грамматическими символами. Если X представляет собой символ, а a – один из его атрибутов, то значение a в некотором узле дерева разбора, помеченном X, записывается как X.a. Если узлы дерева разбора реализованы в виде записей или объектов, то атрибуты X могут быть реализованы как поля данных в записях, представляющих узлы X. Атрибуты могут быть любого вида: числами, типами, таблицами ссылок или строками.

Дерево разбора, в каждом узле которого атрибуты уже вычислены, называется аннотированным (англ. annotated), а процесс вычисления этих атрибутов – аннотированием дерева разбора.

Транслирующий символ – нетерминал, который раскрывается в ε и в момент раскрытия выполняет связанное с ним действие. Действия пишутся в фигурных скобках рядом с транслирующим символом.

Стоит отметить, что не существует гарантии наличия даже одного порядка обхода дерева разбора, при котором вычислятся все атрибуты в узлах.

Атрибут, значение которого зависит от значений атрибутов детей данного узла или от других атрибутов этого узла, то атрибут называется синтезируемым (англ. synthesized attribute).

Грамматика называется S-атрибутной (англ. S-attributed definition), если с атрибутами выполняются только операции присваивания значений других атрибутов, а внутри транслирующих символов происходят обращения только к атрибутам этого транслирующего символа. То есть в грамматике используются только синтезируемые атрибуты. Дерево разбора для такой грамматике всегда может быть аннотировано путем выполнения семантических правил снизу вверх, от листьев к корню.

Атрибут, значение которого зависит от значений атрибутов братьев узла или атрибутов родителя, называется наследуемым (англ. inherited attribute).

Грамматика называется L-атрибутной (англ. L-attributed definition), если значения наследуемых атрибутов зависят только от родителей и братьев слева (то есть не зависят от значений атрибутов братьев справа).

СУО без побочных эффектов иногда называют атрибутной грамматикой. Правила атрибутной грамматики определяют значения атрибутов через значения других атрибутов и констант и не выполняют никаких иных действий.

Дерево разбора с указанием значений атрибутов в каждом узле называется аннотированным (annotated) деревом разбора.

9.2. Синтаксически управляемые схемы трансляции. Транслирующие грамматики

Синтаксически управляемые схемы трансляции представляют собой запись, синтаксически управляемые определения.

Синтаксически управляемая схема трансляции (СУТ) представляет собой контекстно-свободную грамматику с программными фрагментами, внедренными в тела продукций. Эти фрагменты называются семантическими действиями и могут находиться в любой позиции в теле продукции. По соглашению действия располагаются внутри фигурных скобок; фигурные скобки, являющиеся грамматическими символами, заключаются в кавычки.

Любая СУТ может быть реализована путем построения дерева разбора с последующим выполнением действий в порядке в глубину слева направо, т.е. в порядке прямого обхода дерева.

Обычно СУТ реализуется в процессе синтаксического анализа, без построения дерева разбора.

В процессе синтаксического анализа действие в теле продукции выполняется, как только все грамматические символы слева от него сопоставлены входной строке.

СУТ, которые могут быть реализованы в процессе синтаксического анализа, можно охарактеризовать путем вставки вместо действий отличающихся

друг от друга нетерминалов-маркеров; каждый маркер M имеет единственную продукцию M → ε. Если синтаксический анализ грамматики с такими нетерминалами-маркерами может быть выполнен некоторым методом, то СУТ может быть реализована в процессе данного синтаксического анализа.

Основой построения СУ -схем перевода является использование двух грамматик, с помощью которых осуществляется синхронный вывод входной и выходной цепочек. Построение транслирующих грамматик предполагает применение другого подхода, который предусматривает использование одной грамматики и разрешает включение как входных, так и выходных символов в каждое правило такой грамматики.

Транслирующей грамматикой (Т-грамматикой) называется КС-грамматика, множество терминальных символов которой разбито на множество входных символов и множество выходных символов, которые называются также символами действия.

9.3. Преобразователи с магазинной памятью

Магазинные автоматы, рассмотренные в предыдущем разделе, позволяют определить для цепочки, заданной на входе, ее принадлежность к языку, допускаемому автоматом. Настоящий раздел посвящен другому типу моделей устройств, называемому магазинными преобразователями. Подобные устройства позволяют строить по заданной входной цепочки соответствующую ей выходную цепочку. Множество таких пар цепочек называют переводом или трансляцией, допускаемым магазинным преобразователем. Для того чтобы построить преобразователь необходимо заранее знать какой перевод он должен выполнять. Кроме того, преобразователь можно построить не для любого перевода, а только для такого, который может быть описан с помощью простой СУ-схемы. Для построения детерминированного преобразователя необходимо еще, чтобы входная грамматика заданной СУ-схемы порождала детерминированный язык.

Магазинный преобразователь (Мп) отличается от магазинного автомата наличием дополнительной выходной ленты, на которую записывается выходная цепочка.

Преобразователем с магазинной памятью (МП-преобразователем) называется восьмерка $P = (Q, T, Г, П, D, q_0, Z_0, F)$, где все символы имеют тот же смысл, что и в определении МП-автомата, за исключением того, что П – конечный выходной алфавит, а D – отображение множества $Q \times (T \cup \{\varepsilon\}) \times Г$ в множество конечных подмножеств множества $Q \times Г^* \times П^*$.

Определим конфигурацию преобразователя P как четверку (q, x, u, y),

где $q \in Q$ – состояние, $x \in T^*$ – цепочка на входной ленте, $u \in Г^*$ – содержимое магазина, $y \in П^*$ – цепочка на выходной ленте, выданная вплоть до настоящего момента.

Если множество D(q, a, Z) содержит элемент (r, u, z), то будем писать (q, ax, Zw, y) ⊢ (r, x, uw, yz) для любых x ∈ T* , w ∈ Γ* и y ∈ П*. Рефлексивно-транзитивное замыкание отношения ⊢ будем обозначать ⊢*.

Цепочку y назовем выходом для x, если (q0 , x, Z0, e) ⊢* (q, e, u, y) для некоторых q ∈ F и u ∈ Γ* . Переводом (или трансляцией), определяемым МП-преобразователем P (обозначается τ(P)), назовем множество

{(x, y) | (q_0 , x, Z_0, e) ⊢* (q, e, u, y) для некоторых q ∈ F и u ∈ Γ*}

Будем говорить, что МП-преобразователь P является детерминированным (ДМП-преобразователем), если выполняются следующие условия:

1) для всех q ∈ Q, a ∈ T ∪ {e} и Z ∈ Γ множество D(q, a, Z) содержит не более одного элемента,

2) если D(q, e, Z) 6= ∅, то D(q, a, Z) = ∅ для всех a ∈ T .

10. АВТОМАТИЗАЦИЯ ПОСТРОЕНИЯ ТРАНСЛЯТОРОВ

Первые компиляторы появились в начале 1950-х годов. В то время процесс написания компилятора считался крайне сложным, так на создание первого компилятора языка Fortran потребовалось 18 человеко-лет работы. С тех пор много усилий было напревлено на упрощение и автоматизацию процесса написания трансляторов. Например, были успешно автоматизированы процессы написания лексических и синтаксических анализаторов по формальному описанию грамматики.

10.1. Генератор лексических анализаторов Lex

Для построения лексических анализаторов существует множество различных программ. Наиболее известной среди них является программа lex. На вход этой программе подаётся описание лексем с помощью регулярных выражений. Для данной автоматной грамматики lex создаёт код лексического анализатора на некотором языке программирования.

Первоначально этот конструктор лексических анализаторов появился в ОС UNIX и порождал сканеры на Си. Существуют версии и аналоги программы Lex для различных операционных систем и языков программирования. В дальнейшем мы будем рассматривать совместимый с lex конструктор лексических анализаторов flex.

Flex представляет собой переписанную с нуля версию lex из AT&T Unix с некоторыми расширениями и несовместимостями. Flex почтии полностью соответствует стандарту POSIX и во многом совместим с lex. Опция -l включает режим максимальной совместимости с lex.

flex читает описание генерируемого анализатора из данного файла или стандартного потокай ввода, если не было указано имя файла, и как

результат генерирует файл с кодом на Си, который, по умолчанию, называется lex.yy.c. В этом файле определяется функция yylex(), которая представляет собой реализацию описанного сканера. Эта функция читает из входного потока символы и, когда среди них встречается лексема выполняет заданный для этой лексемы код.

Описание сканера состоит из следующих разделов разделённых строкой, содержащей только “%%”:

- раздел определений
- раздел правил
- раздел пользовательского кода

```
Определения
%%
Правила
%%
Пользовательский код
```

Раздел определений состоит из определений имён и начальных условий. Определения имён предназначены для упрощения спецификации анализатора и задаются следующим образом

```
Имя Определение
```

Где Имя – слово, начинающееся с буквы или символа подчёркивания, затем следует ноль или более букв, цифр, символов подчёркивания или дефисов. Определение начинается от первого непробельного символа после имени и продолжается до конца строки. В последующем, на определение можно ссылаться по соответствующему имени: {Имя} -⟶ (Определение).

Непосредственно в выходной файл вне функции yylex(), копируется:

- комментарии (/* …*/), открытые в начале строки;
- строки, начинающиеся с ненулевого количества пробельных символов;
- текс заключенный между %{ и %}, расположенными в начале строк.

Это позволят делать глобальные объявления.

Кроме того, текст заключённый между %top{ и }, расположенными в начале строк, помещается в выходной файл до созданных flex определений.

Раздел правил состоит из последовательности правил, записанных в форме

```
Шаблон Действие
```

Шаблон должен располагаться в самом начале строки. Описание действия должно начинаться на той же строке, где и соответствующий шаблон.

Непосредственно в выходной файл внутри функции yylex(), копируется:

- комментарии, расположенные там, где не ожидается шаблон;
- строки, начинающиеся с ненулевого количества пробельных символов;

- текс заключенный между %{ и %}, расположенными в начале строк. Это позволят делать локальные объявления.

Шаблон записывается с использованием расширенного набора регулярных выражений. Вот некоторые из этих выражений:

- x – символ x
- . – любой символ кроме символа новой строки
- [xyz] – один из символов x, y или z
- [a-f] – один из символов из диапозона, начиная от a заканчивая f
- [^A-Z] – один из символов, не являющихся заглавной латинской буквой
- r* – ноль или более выражений r
- r+ – один или более выражений r
- r? – ноль или одно выражение r
- r{2,5} – от 2 до 5 выражений r
- r{2,} – от 2 выражений r
- r{4} – в точности 4 выражения r
- "[xyz]"\"foo" – строка [xyz]"foo
- (r) – выражение r используется для указания порядка выполнения операций над выражениями
- rs – конкатенация
- r|s – или r, или s
- r/s – выражение r, если за ним следует выражение s (s возвращается во входной поток и будет прочитано повторно)
- ^r – выражение r в начале строки
- r$ – выражение r в конце строки

Шаблон заканчивается непосредственно перед первым неэкранированным пробельным символом. Остаток строки является действием. Действие может быть произвольным оператором языка Си, включая оператор return. Если действие содержит {, то оно продолжается до }, в том числе и на последующих строках.

Раздел пользовательского кода непосредственно копируется в выходной файл. Наличие этого раздела необязательно. Если раздел отсутствует, то и отделяющая его строка %% также может отсутствовать.

Когда сгенерированный сканер запущен, он анализирует ввод, считываемый из глобального входного файла yyin, в поисках строк, соответствующих заданным шаблонам. Если найдено более одного соответствия, то используется соответствие, содержащее больший объем ввода (для выражения r/s размер ввода соответствующий s учитывается, несмотря на то, что он будет возвращён в поток). Если же обоим выражениям соответствует одинаковое количество символов, то используется то выражение, которое задано в разделе правил раньше.

Как только было установлено соответствие между регулярным выражением и вводом, соответствующий текст становится доступным через глобальный указатель yytext, а его длина в глобальной целочисленной переменной yyleng. После этоо выполняется действие соответствующее подходящему шаблону и оставшийся ввод анализируется в поисках очередных совпадений.

Переменная yytext может быть определена двумя способами:

- Указатель на символ (массив символов). В этом случае буфер под совпадение будет увеличиваться динамически, вызов unput() будет уничтожать буфер, и буфер нельзя будет менять внутри действий. Этот режим используется по умолчанию или при указании в разделе объявлений директивы %pointer.
- Массив. В этом случае буфер имеет фиксированную длинну, заданную макросом YYLMAX, вызов unput() не уничтожает буфер, и содержимое буфера можно менять внутри действий, не изменяя длинну строки. Этот режим используется при указании опции -l или директивы %array в разделе объявлений.

После каждого вызова yylex() продолжает анализ с того момента, где он был прерван, до тех пор пока не встретиться конец файла или не будет выполнен return.

Если анализатору встретился конец файла, то его дальнейшее поведение зависит от возвращаемого функцией yywrap() значения:

- если возвращен 0, то анализатор считает, что функция установила новый входной файл, из которого будет продолжено чтение;
- если возвращено ненулевое значение, то сканер останавливается возвращая 0.

Вы должны определить функцию yywrap() или указать в разделе определений директиву %option noyywrap. В последнем случае сканер будет вести себя так, будто функция yywrap() возвращает 1.

Пример Описание генератора, который читает указанный файл, выделяет в нём лексемы и осуществляет аварийную остановку в случае ошибки.

```
%{
#include <stdio.h>
enum TT { MY_EOF = 0, ERROR, NUMBER, IDENT, PUNKT };
%}
%option noyywrap
%%
[0-9]+("."[0-9]+)?([eE][+-]?[0-9]+)? {
return NUMBER; }
[a-zA-Z_][a-zA-Z_0-9]* return IDENT;
"+"|-|"*"|"/"|"("|")"  return PUNKT;
" "                  /* ignore space */
.                       return ERROR;
%%
```

```
int main(int argc, char **argv)
{
    if (argc!=2) {
        printf("Invoke error\n");
        return 1;
    }
    yyin = fopen(argv[1], "rt");
    if (!yyin) {
        printf("Open failed\n");
        return 2;
    }
    enum TT tt;
    while (tt = yylex()) {
        printf("%d\n", tt);
    }
    fclose(yyin);
    return 0;
}
```

10.2. Генератор синтаксических анализаторов YACC

Одним из широко распространённых генераторов синтаксических анализаторов являестя yacc (yet another compiler-compiler). Первая версия yacc была создана С. Джонсоном в начале 1970-х годов. Как и для lex, в дальнейшем для yacc появилось много различных версий. Далее мы будем рассматривать генератор синтаксических анализаторов bison, который конструирует код на Си.

Bison, будучи генератором парсеров общего назначения, преобразует аннотированную контекстно свободную грамматику в соответствующий ей анализатор. Bison является совместимым с yacc в том смысле, что грамматика, корректно подготовленная для yacc будет правильно преобразована bison.

Режим максимальной совместимости с yacc включается опцией -y.

Yacc генерирует файл y.tab.c, который содержит реализацию грамматического анализатора в виде функции yyparse()

Описание грамматики осуществляется в следующем формате.

```
%{
Пролог
%}
Объявления
%%
Правила
%%
Эпилог
```

Пролог содержит код, который будет вставлен до функции yyparse()

Раздел объявлений содержит определения символов используемых при построении грамматики, а также типы данных, которые используются для представления значений.

Существует несколько типов объявлений. Вот некоторые из них:

объявление токена (%token)

- объявление токена с указанием ассоциативности и приоритета (%left, %right, %nonassoc)
- объявление множества используемых типов для представления значений (%union)
- объявление типа, который используется для представления значения нетерминального символа %type
- объявление стартового символа грамматики (%start)

Правила записываются в следующем формате.

```
Нетерминал:  правило1-компоненты...
           | правило2-компоненты...
           ...
           ;
```

Компоненты представляют собой последовательность символов грамматики. Терминальные символы или должны быть предварительно объявлены или представлены в виде строкового литерала. Правило может заканчиваться указанием некоторого действия. Действие представляет собой заключенный в фигурные скобки код на Си с некоторыми дополнениями. Так в выражениях могут использоваться псевдо-идентификаторы $$, $1, $2 и так далее, которые представляют собой переменные, в которых хранятся значения целевого нетерминального символа и 1-го, 2-го и так далее символов правила. Это позволяет установить значение целевого нетерминального символа. Вертикальной чертой разделяются альтернативные правила.

Эпилог непосредственно копируется в результирующий файл.

Сгенерированный парсер для получения очередной лексемы потока вызывает функцию yylex().

При обнаружении синтаксической ошибки происходит вызов функции yyerror()

10.3. Совместное использование Lex и YACC

Для генерации заголовочного файла y.tab.h с определением токенов необходимо вызвать bison с опцией -d.

Значение токена bison получает из глобальной переменной yylval, которая имеет тип определённый макросом YYSTYPE, который, в свою очередь, может быть сгенерирован в заголовочном файле с помощью объявления %union.

На рисунке 1 изображена схема основных аспектов взаимодействия lex и yacc.

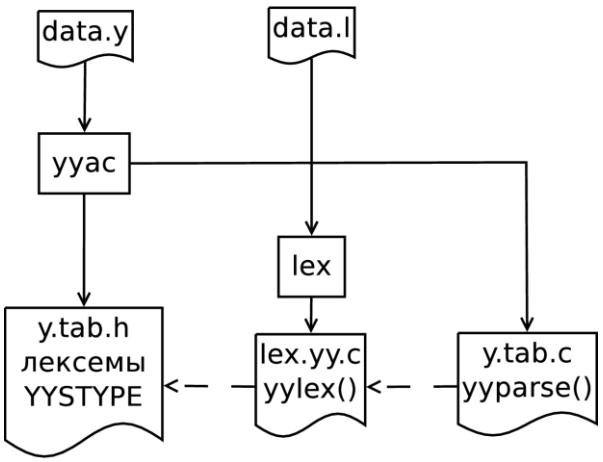


Рисунок 1 – Взаимодействие Lex и YACC

Пример Подсчёт числа лексем. Описание лексического анализатора

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%option noyywrap

%%
[0-9]+("."[0-9]+)?([eE][+-]?[0-9]+)? {
    return NUMBER;
}
[a-zA-Z_][a-zA-Z_0-9]* return IDENT;
"+"|-|"*"|"/"|"("|")"  return PUNKT;
" " /* ignore space */
\n
.                      return ERROR;

%%
int open_lex(char *name)
{
    return (yyin=fopen(name, "rt"))!=NULL;
}
void close_lex()
{
    fclose(yyin);
}
```

Описание синтаксического анализатора

```
%{
#include <stdio.h>
int open_lex(char *);
void close_lex();
```

```
int yylex();
void yyerror(char const *);
%}
%union {
    char * ident;
    double number;
    char punkt;
    int count;
}
%start start
%token <ident> IDENT
%token <number> NUMBER
%token <punkt> PUNKT
%token END 0
%token ERROR
%type <count> list

%%
start: list END { printf("%d\n", $1); }
    ;
list: list element { $$ = $1 + 1; }
    | { $$ = 0; }
    ;
element: IDENT | NUMBER | PUNKT
        ;
%%
void yyerror(char const *msg)
{
    fprintf(stderr, "Error: %s\n", msg);
}

int main(int argc, char **argv)
{
    if (argc!=2) {
        printf("Invoke error\n");
        return 1;
    }
    if (!open_lex(argv[1])) {
        printf("Open failed\n");
        return 1;
    }
    yyparse();
    close_lex();
    return 0;
}
```

11. ПРОМЕЖУТОЧНЫЙ КОД

11.1. Представление в виде ориентированного графа

Простейшей формой промежуточного представления является синтаксическое дерево программы. Ту же самую информацию о входной программе, но в более компактной форме дает ориентированный ациклический граф (ОАГ), в котором в одну вершину объединены вершины синтаксического дерева, представляющие общие подвыражения.

Подобно синтаксическому дереву для выражения ориентированный ациклический граф имеет листья, соответствующие атомарным операндам, и внутренние узлы, соответствующие операторам. Отличие состоит в том, что узел N в ориентированном ациклическом графе имеет более одного родителя, если N представляет собой общее подвыражение; в синтаксическом дереве поддерево для общего подвыражения будет продублировано столько раз, сколько раз это подвыражение встречается в исходном выражении. Таким образом, ориентированный ациклический граф не только представляет выражение более кратко, но и дает генератору кода возможность генерации более эффективного кода для вычисления выражения.

11.2. Инфиксная и постфиксная формы

Инфиксная нотация – это форма записи математических и логических формул, в которой операторы записаны в инфиксном стиле между операндами, на которые они воздействуют (например, 2 + 2). Задача разбора выражений, записанных в такой форме, для компьютера сложнее по сравнению с префиксной (то есть + 2 2) или постфиксной (2 2 +). Однако эта запись используется в большинстве языков программирования как более естественная для человека.

В инфиксной нотации, в отличие от префиксной и постфиксной, скобки, окружающие группы операндов и операторов, определяют порядок, в котором будут выполнены операции. При отсутствии скобок операции выполняются согласно правилам приоритета операторов.

Инфиксная запись может отличаться от функциональной, где имя функции описывает какую-то операцию, а её аргументы являются операндами. Примером функциональной записи может быть S (1, 3) в которой функция S означает операцию сложения: S(1, 3) = 1 + 3 = 4.

Обратная польская запись (англ. Reverse Polish notation, RPN) – форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Также именуется как обратная бесскобочная запись, постфиксная нотация, бесскобочная символика Лукасевича, польская инверсная запись, ПОЛИЗ.

Стековой машиной называется алгоритм, проводящий вычисления по обратной польской записи

Отличительной особенностью обратной польской нотации является то, что все аргументы (или операнды) расположены перед знаком операции. В общем виде запись выглядит следующим образом:

Запись набора операций состоит из последовательности операндов и знаков операций. Операнды в выражении при письменной записи разделяются пробелами.

Выражение читается слева направо. Когда в выражении встречается знак операции, выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их записи. Результат операции заменяет в выражении последовательность её операндов и её знак, после чего выражение вычисляется дальше по тому же правилу.

Результатом вычисления выражения становится результат последней вычисленной операции.

Например, рассмотрим вычисление выражения 7 2 3 * − (эквивалентное выражение в инфиксной нотации: 7 − 2 * 3).

Первый по порядку знак операции – «*», поэтому первой выполняется операция умножения над операндами 2 и 3 (они стоят последними перед знаком). Выражение при этом преобразуется к виду 7 6 − (результат умножения – 6, – заменяет тройку «2 3 *»).

Второй знак операции – «−». Выполняется операция вычитания над операндами 7 и 6.

Вычисление закончено. Результат последней операции равен 1, это и есть результат вычисления выражения.

Очевидное расширение обратной польской записи на унарные, тернарные и операции с любым другим количеством операндов: при использовании знаков таких операций в вычислении выражения операция применяется к соответствующему числу последних встретившихся операндов.

Особенности обратной польской записи следующие:

Порядок выполнения операций однозначно задаётся порядком следования знаков операций в выражении, поэтому отпадает необходимость использования скобок и введения приоритетов и ассоциативности операций.

В отличие от инфиксной записи, невозможно использовать одни и те же знаки для записи унарных и бинарных операций. Так, в инфиксной записи выражение 5 * (−3 + 8) использует знак «минус» как символ унарной операции (изменение знака числа), а выражение (10 − 15) * 3 применяет этот же знак для обозначения бинарной операции (вычитание). Конкретная операция определяется тем, в какой позиции находится знак. Обратная польская запись не позволяет этого: запись 5 3 − 8 + * (условный аналог первого выражения) будет интерпретирована как ошибочная, поскольку невозможно определить, что «минус» после 5 и 3 обозначает не вычитание; в результате будет сделана попытка вычислить сначала 5 − 3, затем 2 + 8, после чего выяснится, что

для операции умножения не хватает операндов. Чтобы всё же записать это выражение, придётся либо переформулировать его (например, записав вместо выражения − 3 выражение 0 − 3), либо ввести для операции изменения знака отдельное обозначение, например, «±»: 5 3 ± 8 + *.

Так же, как и в инфиксной нотации, в ОПН одно и то же вычисление может быть записано в нескольких разных вариантах. Например, выражение (10 − 15) * 3 в ОПН можно записать как 10 15 − 3 *, а можно – как 3 10 15 − *

Из-за отсутствия скобок обратная польская запись короче инфиксной. За счёт этого при вычислениях на калькуляторах повышается скорость работы оператора (уменьшается количество нажимаемых клавиш), а в программируемых устройствах сокращается объём тех частей программы, которые описывают вычисления. Последнее может быть немаловажно для портативных и встроенных вычислительных устройств, имеющих жёсткие ограничения на объём памяти.

11.3. Трехадресный код

Трехадресный код – это последовательность операторов вида x := y op z, где x, y и z – имена, константы или сгенерированные компилятором временные объекты. Здесь op – двуместная операция, например операция плавающей или фиксированной арифметики, логическая или побитовая. В правую часть может входить только один знак операции.

Составные выражения должны быть разбиты на подвыражения, при этом могут появиться временные имена (переменные). Смысл термина "трехадресный код" в том, что каждый оператор обычно имеет три адреса: два для операндов и один для результата. Трехадресный код – это линеаризованное представление синтаксического дерева или ОАГ, в котором временные имена соответствуют внутренним вершинам дерева или графа. Например, выражение x + y * z может быть протранслировано в последовательность операторов

```
t1 := y * z
t2 := x + t1
```

где t1 и t2 - имена, сгенерированные компилятором. В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления программы и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться. Например, условный оператор

```
if A > B then S1 else S2
```

может быть представлен следующим кодом:

```
t := A - B
JGT t, S2
...
```

Здесь JGT – двуместная операция условного перехода, не вырабатывающая результата.

Разбиение арифметических выражений и операторов управления делает трехадресный код удобным при генерации машинного кода и оптимизации. Использование имен промежуточных значений, вычисляемых в программе, позволяет легко переупорядочивать трехадресный код.

Трехадресный код – это абстрактная форма промежуточного кода. В реализации трехадресный код может быть представлен записями с полями для операции и операндов. Рассмотрим три способа реализации трехадресного кода: четверки, тройки и косвенные тройки.

Четверка – это запись с четырьмя полями, которые будем называть op, arg1, arg2 и result. Поле op содержит код операции. В операторах с унарными операциями типа x := -y или x := y поле arg2 не используется. В некоторых операциях (типа "передать параметр") могут не использоваться ни arg2, ни result. Условные и безусловные переходы помещают в result метку перехода.

11.4. Статические единственные присваивания. Проект LLVM

Представление в виде статических единственных присваиваний (СЕП) является промежуточным представлением, которое облегчает некоторые из оптимизаций кода. СЕП отличается от трех адресного кода. Первое отличие заключается в том,что все присваивания в СЕП выполняются для переменных с различными именами; отсюда следует название единственного присваивания.

Одна и та же переменная может быть определена в программе в двух разных путях потока управления. СЕП использует для комбинации двух определений x соглашение о записи, именуемой φ-функцией. Она возвращает значение своего аргумента, соответствующее пути потока управления, который был пройден до команды присваивания с участием φ-функции.

LLVM (ранее Low Level Virtual Machine) – проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. Состоит из набора компиляторов из языков высокого уровня (так называемых «фронтендов»), системы оптимизации, интерпретации и компиляции в машинный код. В основе инфраструктуры используется RISC-подобная платформонезависимая система кодирования машинных инструкций (байт-код LLVM IR), которая представляет собой высокоуровневый ассемблер, с которым работают различные преобразования.

Написан на C++, обеспечивает оптимизации на этапах компиляции, компоновки и исполнения. Изначально в проекте были реализованы компиляторы для языков Си и C++ при помощи фронтенда Clang, позже появились фронтенды для множества языков, в том числе: ActionScript, Ада,

C#, Common Lisp, Crystal, CUDA, D, Delphi, Dylan, Fortran, Graphical G Programming Language, Halide, Haskell, Java (байткод), JavaScript, Julia, Kotlin, Lua, Objective-C, OpenGL Shading Language, Ruby, Rust, Scala, Swift, Xojo.

LLVM может создавать машинный код для множества архитектур, в том числе ARM, x86, x86-64, PowerPC, MIPS, SPARC, RISC-V и других (включая GPU от Nvidia и AMD).

Некоторые проекты имеют собственные LLVM-компиляторы (например LLVM-версия GCC), другие используют инфраструктуру LLVM, например таков Glasgow Haskell Compiler.

Разработка начата в 2000 году в Университете Иллинойса. К середине 2010-х годов LLVM получил широкое распространение в индустрии: использовался, в том числе, в компаниях Adobe, Apple и Google. В частности, на LLVM основана подсистема OpenGL в Mac OS X 10.5, а iPhone SDK использует препроцессор (фронтенд) GCC с бэкэндом на LLVM. Apple и Google являются одними из основных спонсоров проекта, а один из основных разработчиков – Крис Латтнер – 11 лет проработал в Apple (с 2017 года – в Tesla Motors, с 2020 года – в разработчике процессоров и микроконтроллеров на архитектуре RISC-V SiFive).

В основе LLVM лежит промежуточное представление кода (Intermediate Representation, IR), над которым можно производить трансформации во время компиляции, компоновки и выполнения. Из этого представления генерируется оптимизированный машинный код для целого ряда платформ, как статически, так и динамически (JIT-компиляция). LLVM 9.0.0 поддерживает статическую генерацию кода для x86, x86-64, ARM, PowerPC, SPARC, MIPS, RISC-V, Qualcomm Hexagon, NVPTX, SystemZ, Xcore. JIT-компиляция (генерация машинного кода во время исполнения) поддержана для архитектур x86, x86_64, PowerPC, MIPS, SystemZ, и частично ARM.

LLVM написана на C++ и портирована на большинство Unix-подобных систем и Windows. Система имеет модульную структуру, отдельные её модули могут быть встроены в различные программные комплексы, она может расширяться дополнительными алгоритмами трансформации и кодогенераторами для новых аппаратных платформ.

В LLVM включена обёртка API для OCaml.

12. ОПТИМИЗАЦИЯ

12.1. Основные источники оптимизации

Оптимизация, выполняемая компилятором, должна сохранять семантику исходной программы. За исключением очень редких случаев, если программист выбрал и реализовал конкретный алгоритм, компилятор не в состоянии достаточно хорошо разобраться в программе, чтобы заменить его совершенно иным более эффективным алгоритмом. Компилятор знает только о том, как применять относительно низкоуровневые семантические преобразования с использованием обобщенных фактов, таких как алгебраические тождества наподобие i + 0 = i или семантика программы наподобие того факта, что выполнение одинаковых операций над одинаковыми значениями даст одинаковые результаты.

В типичной программе имеется масса избыточных операций. Иногда избыточность проявляется на уровне исходного текста программы. Например,программист может счесть более удобным вычислить некоторый результат заново,оставляя компилятору распознать, что одно из вычислений излишне. Но более часто избыточность оказывается побочным действием написания программ на языке программирования высокого уровня. В большинстве языков программирования (отличных от C и C++, в которых разрешены арифметические действия с указателями), у программиста нет выбора вариантов обращения к элементам массива или полям структуры.

При компиляции программы каждое из таких обращений к высокоуровневым структурам данных разворачивается в ряд низкоуровневых арифметических операций, таких как вычисление (r, j)-го элемента матрицы A. Обращения к одной и той же структуре данных часто используют много одинаковых низкоуровневых операций. Программист не осведомлен об этих операциях и не может устранить их избыточность самостоятельно. Более того, с точки зрения инженерии программного обеспечения предпочтительно, чтобы программист обращался к элементам данных только по их высокоуровневым именам; такие программы проще писать и, что более важно, проще понимать и развивать. Обладая компилятором с устранением избыточности, мы получаем лучшее из двух миров – программы оказываются одновременно эффективными и легко поддерживаемыми.

Существует ряд способов, которыми компилятор может улучшить программу без изменения вычисляемой функции. Устранение общих подвыражений,размножение копий, удаление недоступного кода и дублирование констант – вот основные примеры таких преобразований, сохраняющих функции (или сохраняющих семантику (semantics-preserving)).

12.2. Анализ потока данных

Под анализом потоков данных понимают совокупность задач, нацеленных на выяснение некоторых глобальных свойств программы, то есть извлечение информации о поведении тех или иных конструкций в некотором контексте. Такая постановка задачи возможна по той причине, что язык программирования и вычислительная среда определяют некоторую общую, "безопасную" семантику конструкций, которая годится "на все случаи жизни". Учет же контекстных условий позволяет делать более конкретные, частные заключения о поведении той или иной конструкции; при этом такие заключения, вообще говоря, перестают быть верными в другом контексте. Например, общая семантика присваивания заключается в вычислении выражения, стоящего в правой части, и присваивании полученного значения в переменную, стоящую в левой части. Однако в случае, когда выражение в правой части не имеет побочных эффектов, а переменная в левой части более нигде не используется, данный оператор становится эквивалентен пустому.

Для того чтобы описать понятие контекста, снова обратимся к графу потока управления (см. "Оптимизация" и "Анализ потока управления"). Понятно, что на смысл каждой конструкции может оказывать влияние любая конструкция, из которой в этом графе достижима данная. Отсюда следует, что для правильного учета контекста необходимо учесть влияние всех путей до данной вершины, сначала определив влияние каждого пути, а затем выделив общую часть. Задача осложняется тем, что при наличии контуров множество всех путей в графе управления становится бесконечным.

12.3. Распространение констант

Свёртка констант (англ. constant folding) и распространение констант (так же продвижение констант, дублирование констант, англ. constant propagation) – часто используемые в современных компиляторах оптимизации, уменьшающие избыточные вычисления, путём замены константных выражений и переменных на их значения. Так же часто применяется расширенный алгоритм sparse conditional constant propagation, выполняющий одновременно распространение констант и удаление некоторого мёртвого кода.

Свёртка констант – оптимизация, вычисляющая константные выражения на этапе компиляции. Прежде всего, упрощаются константные выражения, содержащие числовые литералы. Также могут быть упрощены выражения, содержащие никогда не изменяемые переменные или переменные, объявленные как константы. Рассмотрим пример:

```
i = 320 * 200 * 32;
```

Компилятор, поддерживающий свёртку констант, не будет генерировать две инструкции умножения и запись полученного результата. Вместо этого он распознает эту конструкцию как константное выражение и заменит её на вычисленное значение (в данном случае 2 048 000).

Распространение констант – оптимизация, заменяющее выражение, которое при выполнении всегда возвращает одну и ту же константу, самой этой константой. Это может быть константа, определённая ранее, или встроенная функция, применённая к константам.

12.4. Устранение частичной избыточности

Выполняя перемещения кода и при необходимости сохраняя результат во временной переменной, зачастую можно снизить количество вычислений такого выражения вдоль многих путей выполнения, при этом ни по одному из возможных путей количество вычислений не увеличится. Заметим, что количество различных мест,где выполняется вычисление x + y, может и увеличиться, но это относительно неважно, поскольку снизится количество вычислений выражения x + y.

Избыточность в программе существует в различных формах. Она может быть в виде общих подвыражений, когда несколько вычислений выражения дают одно и то же значение. Она может быть и в виде выражения, инвариантного относительно цикла, которое вычисляет одно и то же значение на каждой итерации цикла. Избыточность может быть частичной, если она обнаруживается вдоль некоторых, но не всех, путей. Общие подвыражения и выражения, инвариантные относительно цикла, можно рассматривать как частные случаи частичной избыточности; таким образом, для устранения различных видов избыточности можно разработать единый алгоритм устранения частичной избыточности.

12.5. Циклы в графах потоков

Важность циклов не подлежит сомнению, по крайней мере потому, что программы затрачивают основное время выполнения именно на работу циклов, так что оптимизация, повышающая производительность циклов, может оказать существенное влияние на производительность программы в целом. Таким образом, желательно четко идентифицировать циклы и рассматривать их отдельно от других видов потоков управления. Циклы также влияют на время работы анализа программы. Если в программе нет ни одного цикла, то ответ на задачи потоков данных можно получить путем одного прохода по программе. Например, задача потока данных в прямом направлении может быть решена путем однократного посещения всех узлов в топологическом порядке.

13. ГЕНЕРАЦИЯ ЦЕЛЕВОГО КОДА

13.1. Общие принципы генерации кода

Последней стадией нашей модели компиляции является генератор кода. Он получает на вход промежуточное представление исходной программы от начальной стадии компилятора и выводит эквивалентную целевую программу.

К генератору кода предъявляются жесткие требования. Получаемый код должен сохранять семантическое значение исходной программы и быть высококачественным, что означает эффективное использование доступных ресурсов целевой машины. Кроме того, эффективно должен работать и сам генератор кода. Математически проблема генерации оптимальной целевой программы для данной исходной является неразрешимой; многие из подзадач, встречающихся при генерации кода, таких как распределение регистров, вычислительно трудноразрешимы. На практике мы вынуждены довольствоваться эвристическими методами, генерирующими хороший, но не обязательно оптимальный код. К счастью, эти эвристики достаточно стары и проверены, так что тщательно разработанный генератор может давать код в несколько раз более быстрый, чем получаемый от простейшего генератора без их применения.

Компиляторы, которые должны давать эффективные целевые программы, перед началом генерации кода включают фазу оптимизации. Оптимизатор превращает одно промежуточное представление в другое, из которого может быть сгенерирован более эффективный код. В общем случае фазы оптимизации и генерации кода, известные как заключительная стадия компилятора, могут выполнять несколько проходов по промежуточному представлению перед генерацией целевой программы.

Перед генератором кода стоят три основные задачи: выбор команд, распределение и назначение регистров и упорядочение команд. Выбор команд означает выбор машинных команд целевой машины для реализации инструкций промежуточного представления. Распределение и назначение регистров означает принятие решения о том, какие значения в каких регистрах будут храниться. Упорядочение команд предусматривает принятие решения о том, в каком порядке должны выполняться сгенерированные команды.

Такие задачи, как выбор команд, распределение регистров и упорядочение команд, стоят практически перед всеми генераторами кода, в то время как их конкретные детали зависят от промежуточного представления, целевого языка и системы времени выполнения. Наиболее важным критерием оценки генератора кода является корректность получающегося кода. Корректность приобретает особую важность в связи с наличием массы частных случаев, с которыми может столкнуться генератор. При сверхприоритетности корректности получающегося кода генераторы кода должны проектироваться так, чтобы их можно было легко реализовывать, тестировать и поддерживать.

13.2. Модели целевой машины

Одним из непременных требований к построению хорошего генератора кода является близкое знакомство с целевой машиной и ее набором инструкций. К сожалению, при обсуждении общих вопросов генерации кода невозможно описать нюансы той или иной целевой машины достаточно подробно, чтобы иметь возможность генерировать для нее хороший код.

Модель нашего целевого компьютера представляет собой трехадресную машину с операциями загрузки и сохранения регистров, вычислительными операциями, условными и безусловными переходами. Компьютер представляет собой машину с байтовой адресацией памяти с п регистрами общего назначения R0, …, Rn–1. Полнофункциональный язык ассемблера должен иметь множество команд. Чтобы не потерять базовые концепции за мириадами несущественных деталей, мы будем использовать очень ограниченное множество команд и считать, что все операнды представляют собой целые числа. Большинство команд состоит из оператора, за которым следуют приемник и список исходных операндов. Команде может предшествовать метка. Предполагается, что имеются команды следующих видов.

Операции загрузки.

Операция сохранения.

Вычислительные операции.

Безусловные переходы.

Условные переходы.

Наша целевая машина имеет несколько режимов адресации.

В командах адрес может быть именем переменной x, что означает место в памяти, зарезервированное для x.

Адрес может быть индексированным адресом вида a (r), где a –переменная, а r – регистр. Адрес a (r) вычисляется путем прибавления к I-значению a значения из регистра r.

Адрес может быть индексирован регистром.

Целевая машина допускает два режима косвенной адресации: *r означает ячейку памяти, находящуюся по адресу, представленному содержимым регистра r, а *100(r) означает ячейку памяти, находящуюся по адресу, полученному путем прибавления 100 к содержимому r.

Наконец, имеется режим адресации с использованием констант (для указания которых используется префикс #).

13.3. Адреса в целевом коде

Каждая выполняющаяся программа работает в собственном логическом адресном пространстве, которое разбивается на четыре области для кода и данных.

1. Статически определенная область Code, в которой хранится выполнимый целевой код.
2. Статически определенная область Static, в которой хранятся глобальные константы и иные данные, генерируемые компилятором. Размер глобальных констант и данных компилятора также может быть определен в процессе компиляции.
3. Динамически управляемая область памяти Heap для хранения объектов данных, память для которых выделяется и освобождается в процессе выполнения программы. Размер области Heap не может быть определен вовремя компиляции.
4. Динамически управляемая область памяти Stack для хранения записей активации (структуры данных создаваемые для каждого вызова подпрограмм, содержат необходимою информацию для выполнения вызова и возврата управления) между их созданием и уничтожением в процессе вызова процедури возвратов из них. Как и в случае кучи Heap, размер области Stack не может быть определен во время компиляции.

13.4. Простой алгоритм генерации кода

Рассмотрим алгоритм генерации кода для отдельного базового блока. Он поочередно рассматривает трехадресные команды и отслеживает,какие значения находятся в регистрах, так что позволяет избежать излишних загрузок и сохранений.

Одним из основных вопросов в процессе генерации кода является принятие решения о том, как наилучшим образом использовать регистры. Существует четыре основных применения регистров.

- В большинстве архитектур некоторые или все операнды должны находиться в регистрах для выполнения операции.
- Регистры представляют собой хорошие временные переменные – места для хранения результатов подвыражений при вычислениях больших выражений или, в общем случае, для размещения переменных, использующихся в пределах только одного базового блока.
- Регистры используются для хранения (глобальных) значений, которые вычисляются в одном базовом блоке и используются в других базовых блоках,например, индекса цикла, который увеличивается на каждой итерации цикла и неоднократно используется в пределах цикла.
- Регистры зачастую используются для помощи в управлении памятью времени выполнения, например для управления стеком времени выполнения,включая поддержку указателя стека и, возможно, элементов на вершине стека.

Это конкурирующие использования регистров, поскольку их количество весьмаограничено. Рассматриваемый алгоритм предполагает наличие некоторого множества регистров, доступных для хранения использующихся в блоке значений. Обычно это множество не включает в себя все имеющиеся в машине регистры, поскольку некоторые из них зарезервированы для глобальных переменных и управления стеком. Мы считаем, что базовые блоки уже преобразованы в идеальные последовательности трехадресных команд при помощи таких трансформаций, как объединение общих подпоследовательностей и др. Мы также предполагаем, что для каждого оператора имеется ровно одна машинная команда, которая получает необходимые операнды в регистрах и выполняет операцию, причем ее результат также находится в регистре.

Алгоритм генерации кода поочередно рассматривает каждую трехадресную команду и определяет, какие загрузки необходимо выполнить, чтобы требуемые операнды находились в регистрах. После генерации загрузок генерируется сама операция, а затем при необходимости сохранения результата в памяти – команды сохранения.

13.5. Распределение и назначение регистров

Команды, включающие в качестве операндов только регистры, выполняются быстрее, чем аналогичные команды с ячейками памяти в качестве операндов. На современных машинах эта разница в скорости может достигать порядка по величине. Следовательно, эффективное использование регистров – жизненно важная составляющая генерации хорошего целевого кода.

Для этого надо рассмотреть различные стратегии принятия решения о том, какие значения в программе должны находиться в регистрах (распределение регистров) и в каком регистре должно храниться каждое значение (назначение регистров).

Один из подходов к распределению и назначению регистров состоит в выделении конкретных регистров для определенных значений в целевой программе. Например, решение может состоять в назначении базовых адресов одной группе регистров; арифметических вычислений – второй; вершины стека времени выполнения – некоторому фиксированному регистру и т. д. Преимуществом такого подхода является упрощение разработки генератора кода. Недостаток же состоит в том, что слишком строгое следование правилу ограничивает эффективность использования регистров; ряд регистров на некотором участке кода может оставаться неиспользованным, в то время как из-за нехватки других регистров будут производиться излишние обмены значениями между регистрами и памятью. Тем не менее в большинстве вычислительных сред имеет смысл зарезервировать несколько регистров в качестве базовых, указателя стека и других и позволить компилятору использовать остальные регистры по своему усмотрению.

13.6. Локальная оптимизация

В то время как большинство промышленных компиляторов дают хороший код путем тщательного выбора команд и распределения регистров, некоторые компиляторы используют иную стратегию: они генерируют простейший код, а затем повышают его качество, применяя "оптимизирующие" преобразования целевой программы. Термин "оптимизирующий" в данном случае несколько неверен,поскольку не гарантируется, что полученный в результате код будет более оптимален с той или иной точки зрения. Тем не менее многие простые преобразования могут существенно улучшить время работы или размер целевой программы.

Простым, но эффективным методом локального улучшения целевого кода является локальная оптимизация (peephole optimization), которая выполняется путем перемещения по целевой программе окна ("peephole" – "глазка") и изучения команд в его пределах с дальнейшей заменой последовательностей команд более быстрыми или более короткими там, где это возможно. Локальная оптимизация зачастую применима непосредственно после генерации промежуточного кода для усовершенствования промежуточного представления. Глазок представляет собой небольшое перемещающееся по программе окно. Код в этом окне не обязательно непрерывен, хотя некоторые реализации и выдвигают такое требование. Характерным для локальной оптимизации является то, что каждое улучшение кода может создать условия для дополнительных улучшений. Вообще говоря, для получения максимального эффекта необходимы повторяющиеся проходы по целевому коду.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Молчанов А.Ю. Системное программное обеспечение. – Санкт-Петербург [и др.]: Питер, 2003. – 395 с.
2. Опалева Э.А. Языки программирования и методы трансляции. – Санкт-Петербург: БХВ-Петербург, 2005. – 476 с.
3. Криницкий Н.А. Алгоритмы вокруг нас. – 2-е изд. – Москва: Наука, 1984. – 223 с.
4. Кузнецов О.П. Дискретная математика для инженера. – 2-е изд., перераб. и доп. – Москва: Энергоатомиздат, 1988. – 480 с.
5. Гладкий А.В. Формальные грамматики и языки. – Москва: Наука, 1973. – 368 с.
6. Гладкий А.В. Элементы математической лингвистики. – Москва: Наука, 1969. – 192 с.
7. Грис, Д. Конструирование компиляторов для цифровых вычислительных машин: пер. с англ. / Д. Грис; под ред. Ю.М. Баяковского, В.С. Штаркмана. – Москва: Мир, 1975. – 544 с.
8. Молчанов А.Ю. Системное программное обеспечение: лабораторный практикум. – Санкт-Петербург [и др.]: Питер, 2005. – 284 с.
9. Касьянов В.Н. Методы построения трансляторов / отв. ред. А.П. Ершов; АН СССР, Сибирское отд-ние, Вычислит. центр. – Новосибирск: Наука, 1986. – 344 с.
10. Ингерман П.З. Синтаксически ориентированный транслятор / пер. с англ. Ф.Ф. Шиллер; под ред. [и с предисл.] А.И. Китова. – Москва: Мир, 1969. – 174 с.
11. Мартыненко Б.К. Языки и трансляции: учеб. пособие. – Изд. 2-е, испр. и доп. – СПб.: Изд-во С.-Петерб. ун-та, 2008. – 257 с.
12. Дуванов А.А. Транслятор?.. Это очень просто! – Москва: Чистые пруды, 2009. – 32 с.
13. Серебряков, В.А. Основы конструирования компиляторов / В.А. Серебряков, М.П. Галочкин. – Москва, 1999. – 175 с.
14. Компиляторы: принципы, технологии и инструментарий / А.В. Ахо, М.С. Лам, Р. Сети, Дж.Д. Ульман. – 2-е изд. – М.: ООО «И.Д. Вильямс», 2008. – 1184 с.
15. Введение в теорию автоматов, языков и вычислений / Д.Э. Хопкрофт, Р. Мотвани, Дж.Д. Ульман. – 2-е изд. – М.: Издательский дом «Вильямс», 2008. – 528 с.
16. Свердлов С.З. Языки программирования и методы трансляции. – СПб.: Питер, 2007. – 638 с.
17. Хартер Р. Проектирование и конструирование компиляторов. – М.: Финансы и статистика, 1984. – 232 с.
18. Пирс Б. Типы в языках программирования. – М.: Издательство «Лямбда пресс»: «Добросвет», 2011. – 656 с.
19. Вирт Н. Алгоритмы+структуры данных=программы. – М.: Мир, 1985. – 406 с.
20. Себеста Р.У. Основные концепции языков программирования. – 5-е изд. – М.: Издательский дом «Вильямс», 2001. – 672 с.
21. Свердлов С.З. Конструирование компиляторов. – Lambert Academic Publishing, 2015. – 575 с.
22. Карпов Ю.Г. Теория и технология программирования. Основы построения трансляторов. – СПб.: БХВ-Петербург, 2005. – 271 с.

Учебное издание

ТРАНСЛЯЦИЯ ФОРМАЛЬНЫХ ЯЗЫКОВ

Курс лекций

Составитель
СЕРГЕЕНКО Сергей Владимирович

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *Л.В. Рудницкая*

Подписано в печать 28.08.2023. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 5,23. Уч.-изд. л. 4,82. Тираж 40 экз. Заказ 84.

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.