Автоматическая генерация позитивных и негативных тестов для тестирования фазы синтаксического анализа

С.В. Зеленов, С.А. Зеленова

Аннотация. В статье описывается методика автоматической генерации наборов позитивных и негативных тестов для тестирования фазы синтаксического анализа. Предлагаются критерии покрытия для таких наборов, основанные на модельном подходе к тестированию, и методы генерации наборов тестов, удовлетворяющих предложенным критериям. Также приводятся результаты практического применения описанной методики к тестированию синтаксических анализаторов различных языков, в том числе языков C и Java.

1. Введение

Компилятор является инструментом, требования к надежности которого чрезвычайно высоки. И это неудивительно, ведь от правильности работы компилятора зависит правильность работы всех скомпилированных им программ. Из-за сложности входных данных и преобразований задача тестирования компиляторов является весьма трудоемкой и непростой. Поэтому вопрос автоматизации всех фаз тестирования (создания тестов, их прогона, оценки полученных результатов) стоит здесь особенно остро.

Синтаксический анализ является частью функциональности любого компилятора. От корректности синтаксического анализа зависит корректность практически всей остальной функциональности – проверки семантических ограничений, оптимизирующих преобразований, генерации кода. Поэтому решение задачи тестирования синтаксических анализаторов является базой для решения задач тестирования всех остальных компонент компилятора.

Для очень многих языков программирования существует формальное описание синтаксиса – описание грамматики языка в форме BNF, а для тех языков, для которых существуют только эталонные компиляторы (например, COBOL), делаются активные попытки построить такое описание (см. [9, 10, 14]). BNF языка является одновременно и спецификацией функциональности синтаксического анализа, таким образом, в этой области наиболее привлекательным является тестирование на основе спецификаций (см. [17]). Существование формального описания позволяет автоматизировать процесс

построения тестов, что существенно снижает трудозатраты, а систематичность тестирования повышает доверие к его результатам.

Построением тестов по грамматике занимались многие авторы. Основополагающей работой в этой области является работа [18], в которой сформулирован следующий критерий покрытия для множества позитивных тестов: для каждого правила в данной грамматике в множестве тестов должно присутствовать предложение языка, в выводе которого используется это правило. В той же работе Пардом предложил метод построения минимального тестового набора, удовлетворяющего этому критерию. Однако указанный критерий оказался недостаточным. Ламмель в работе [9] показал, что тестовые наборы, построенные алгоритмом Пардома, не обнаруживают простейших ошибок. Ламмель также предложил более сильный критерий покрытия, состоящий в том, что покрывается каждая пара правил, одно из которых можно применить непосредственно после другого.

Предлагаемые другими авторами методы являются вероятностными (см. [7, 13, 11, 12]) и не описывают критериев покрытия, и потому для них возникает вопрос остановки генерации тестов, который решается, например, с помощью введения вероятностей появления правил и уменьшения этих вероятностей при каждом новом появлении правила в выводе. В любом случае завершение работы алгоритма за конечное время является проблемой. Кроме того, произвольность остановки генерации нарушает систематичность тестирования.

Все приведенные выше работы касаются генерации позитивных тестов для синтаксического анализатора (т.е. тестов, являющихся предложениями целевого языка). В настоящее время работы, предлагающие методы генерации негативных тестов для синтаксических анализаторов (т.е. тестов, не принадлежащих целевому языку), практически отсутствуют. Однако такие тесты также важны, поскольку пропуск неверной последовательности лексем на этапе синтаксического анализа может привести к аварийному завершению компиляции.

В работе [8] высказано предположение, что если имеется генератор предложений языка из грамматики (генератор позитивных тестов для синтаксического анализатора), то для генерации негативных тестов для синтаксического анализатора можно использовать метод мутационного тестирования (mutation testing)[1] (см. [6, 15]). Идея состоит в том, что в

[1] В общих словах мутационное тестирование состоит в следующем. Из тестируемого компонента получают множество его модификаций (мутантов), каждая из которых содержит ровно одну ошибку. Говорят, что мутант *убит*, если на некоторых входных данных его выход отличен от выхода исходной программы. Если имеется множество тестовых входных данных для программы, то с помощью мутационного анализа (mutation analysis) можно оценить качество этих тестов. Именно, если имеется множество мутантов, то критерий покрытия говорит, что все мутанты должны быть убиты. В случае синтаксического анализатора логично в качестве мутируемого материала рассматривать грамматику языка, т.к. ошибочный анализатор фактически распознает другой язык. В этом случае грамматика-мутант будет убита, если найдется последовательность токенов, принадлежащая языку, задаваемому этой грамматикой-мутантом и не принадлежащая

(продолжение на следующей стр.)

исходную грамматику вносятся изменения (мутации) для получения грамматик, описывающих близкие, но не эквивалентные исходному языки. Эти мутированные грамматики подаются на вход генератору тестов для получения потенциально негативных тестов. Общие проблемы данного подхода состоят в следующем:

- Грамматика-мутант может оказаться эквивалентной исходной грамматике. Такие мутанты должны быть выявлены и не должны использоваться для генерации тестов.
- Даже если грамматика-мутант не эквивалентна исходной, полученные из нее тесты могут оказаться правильными. Выявить эти тесты можно лишь прогнав их через эталонный синтаксический анализатор, которого может и не быть (например, в случае создания нового или расширения существующего языка).

В настоящей работе описаны критерии покрытия, нацеленные на алгоритмы синтаксического анализа. Такой подход представляется оправданным, поскольку тестовые наборы строятся для тестирования синтаксических анализаторов, и эффективность тестового набора должна оцениваться исходя из характеристик, относящихся к тестируемым компонентам (т.е. синтаксическим анализаторам), таких как, например, покрытие функциональности или кода. Данная методика разработана в рамках общего модельного подхода к тестированию компиляторов (см. [2, 3, 4]). Мы рассматриваем известные алгоритмы синтаксического анализа в качестве алгоритмов, моделирующих поведение синтаксического анализатора. Как уже говорилось, в литературе практически отсутствуют работы, посвященные генерации негативных тестов. Настоящая работа призвана закрыть этот пробел.

Статья состоит из введения и трех разделов. В первом разделе содержатся сведения из теории алгоритмов синтаксического анализа. Второй раздел посвящен описанию предлагаемой методики. В нем вводятся понятия позитивных и негативных тестов, описываются критерии покрытия для тестовых наборов, опирающиеся на алгоритмы синтаксического анализа, а также приводятся алгоритмы построения наборов, удовлетворяющих этим критериям покрытия. В третьем разделе описаны результаты практического применения методики.

2. Предварительные сведения

В этом разделе мы приводим некоторые сведения из теории синтаксического анализа. Более подробное изложение приведенных фактов можно найти в известной книге А. Ахо, Р. Сети и Д. Ульмана (см. [5]).

исходному языку (в терминах анализаторов это как раз будет означать, что анализатор-мутант распознал данное предложение, а исходный анализатор - нет, т.е. анализатор-мутант оказался убитым).

Грамматика формального языка задается четверкой $G = (\mathsf{T},\mathsf{N},\mathsf{P},S)$, где

- T – множество терминальных символов или токенов;
- N – множество нетерминальных символов;
- P – список правил грамматики;
- S – стартовый символ грамматики.

Множество предложений формального языка, задаваемого грамматикой G, будем обозначать $\mathfrak{L}_G$.

Дадим несколько определений.

Расширением грамматики G (или просто расширенной грамматикой) называется грамматика $G' = (\mathsf{T},\mathsf{N}',\mathsf{P}',S')$, где S' – новый нетерминальный символ, а к множеству правил добавлено правило $S' \to S$.

Сентенциальной формой будем называть последовательность грамматических символов – нетерминалов и токенов. Далее, греческими буквами из начала алфавита (α, β, γ, δ, ...) мы будем обозначать какие-либо сентенциальные формы. Пустую сентенциальную форму будем обозначать через ε.

Правосентенциальной формой называется сентенциальная форма, для которой существует правый вывод из стартового правила.

Пример. Рассмотрим следующую грамматику:

$$S \to AB$$
$$A \to cd$$
$$B \to eCf$$
$$C \to Ae$$

В ней сентенциальная форма cdB не имеет правого вывода, т.е. не может быть получена с помощью последовательного раскрытия самых правых нетерминалов. Примером правосентенциальной формы может служить форма $AeCf$. ►

Основой правосентенциальной формы называется подпоследовательность ее символов, которая может быть свернута в некоторый нетерминал, такая, что сентенциальная форма, полученная из исходной после свертки, может быть свернута в стартовый символ.

Пример. Пусть грамматика та же, что и в предыдущем примере. Форма $AeCf$, как мы уже заметили, является правосентенциальной. В ней есть две подпоследовательности символов Ae и eCf, которые могут быть свернуты в нетерминалы C и B соответственно. Однако основой является только последовательность eCf, так как сентенциальная форма CCf невыводима из стартового символа. ►

Активным префиксом правосентенциальной формы называется префикс, не выходящий за границы самой правой основы этой формы.

Пунктом грамматики G называется правило вывода с точкой в некоторой позиции правой части. Множество всех пунктов грамматики G будем обозначать через $\mathfrak{P}$.

Пример. Правило вывода $A \to XYZ$ дает 4 пункта: $A \to \bullet XYZ$, $A \to X\bullet YZ$, $A \to XY\bullet Z$ и $A \to XYZ\bullet$. ►

Базисным называется пункт с точкой не у левого края правой части правила, а также пункт $S' \to \bullet S$ в расширенной грамматике.

Замыканием множества пунктов I (обозначается $closure(I)$) называется наименьшее множество пунктов, содержащее I в качестве подмножества такое, что для всякого пункта $A \to \alpha\bullet B\beta \in I$ и любого правила $B \to \gamma$ пункт $B \to \bullet\gamma$ лежит в $closure(I)$.

Для пары (I,X), где I некоторое множество пунктов грамматики G, а X – символ грамматики (терминал или нетерминал), определим функцию $goto(I,X)$ – замыкание множества всех пунктов $A \to \alpha X\bullet\beta$ таких, что $A \to \alpha\bullet X\beta \in I$.

Рассмотрим расширенную грамматику $G' = (\mathsf{T},\mathsf{N}',\mathsf{P}',S')$, где $\mathsf{N}' = \mathsf{N} \cup \{S'\}$, $\mathsf{P} = \mathsf{P} \cup \{S' \to S\}$. Пусть $I_0 = closure(\{S' \to \bullet S\})$. Начиная с I_0, строится система множеств пунктов $I_0,...,I_N$ так, что для всякой пары (I_k,X), где $k = 0,...,N$ и X – символ грамматики, существует индекс $j = 0,...,N$ такой, что $goto(I_k,X) = I_j$. Эта система пунктов называется *канонической системой множеств пунктов*. Используя каноническую систему $I_0,...,I_N$, можно построить конечный автомат V, распознающий активные префиксы, если в качестве состояний s_j взять канонические множества I_j, а переходы задать с помощью функции $goto$.

Широко известны два класса алгоритмов синтаксического анализа: LL-анализ и LR-анализ (см. [5]). LL-анализатор с помощью диаграммы переходов или таблицы разбора строит левый вывод предложения целевого языка. Нерекурсивная реализация LL-анализатора использует стек и таблицу разбора. Изначально в стеке находится символ конца строки \$ и стартовый символ грамматики. На каждом шаге рассматривается символ на вершине стека X и текущий входной символ a. Действия анализатора определяются этими двумя символами:

- если $X = a = \$$, то анализатор прекращает работу и сообщает об успешном завершении разбора;
- если $X = a \neq \$$, анализатор удаляет из стека символ X и переходит к следующему символу входного потока;
- если X является нетерминалом, анализатор ищет такую альтернативу раскрытия символа X, для которой символ a является допустимым первым символом. После того, как требуемая альтернатива найдена, символ X в стеке заменяется обратной последовательностью символов альтернативы. Например, если искомая альтернатива $X \to ABC$, то анализатор заменит X на вершине стека на последовательность CBA, т.е. на вершине стека

окажется символ A. Конфликты, возникающие в процессе поиска альтернатив, могут разрешаться, например, с помощью “заглядывания вперед”, т.е. просмотра нескольких входных символов вместо одного.

Анализатор завершает работу, когда на вершине стека оказывается символ конца строки \$.

Рассмотрим теперь LR-анализатор, построенный на основе стека. У такого LR-анализатора имеются две основные операции:

- перенос символа из входного потока в стек;
- свертка нескольких последовательных символов на вершине стека в некоторый нетерминал.

Работа анализатора происходит так, что в стеке все время находится активный префикс некоторой правосентенциальной формы. При переносе символа и свертке на вершину стека кладется символ состояния s_j конечного автомата V, кодирующий текущий активный префикс. LR-анализатор принимает решение о переносе или свертке, исходя из пары (символ s_j, текущий токен входного потока). Анализатор завершает работу, когда в стеке оказывается стартовый символ грамматики.

3. Описание методики

3.1. Позитивные и негативные тесты для синтаксического анализатора

В данной работе *парсером* мы называем булевскую функцию, заданную на множестве последовательностей токенов и принимающую значение “истина”, если последовательность является предложением данного формального языка, и “ложь” – иначе. Конечно, реальные парсеры могут иметь дополнительную функциональность (например, помимо булевского значения выдавать дерево разбора или идентификацию ошибки), но здесь мы такую функциональность не рассматриваем.

Позитивный тест для парсера – это последовательность токенов, на которой парсер выдает вердикт “истина”, т.е. последовательность токенов, являющаяся предложением целевого языка.

Негативный тест для парсера – это последовательность токенов, на которой парсер выдает вердикт “ложь”, т.е. последовательность токенов, не являющаяся предложением целевого языка.

Для построения какого-нибудь позитивного теста достаточно, следуя правилам грамматики и ограничив рекурсию правил, вывести из стартового символа некоторую сентенциальную форму, состоящую из одних токенов. При построении же негативных тестов возникают два вопроса: насколько произвольна должна быть соответствующая последовательность токенов, и как добиться того, чтобы она действительно не принадлежала целевому языку.

Сначала ответим на второй вопрос: как получить последовательность токенов, гарантированно не принадлежащую множеству предложений целевого языка.

Рассмотрим грамматику $G = (\mathsf{T}, \mathsf{N}, \mathsf{P}, S)$. Для каждого грамматического символа $X \in \mathsf{T} \cup \mathsf{N}$, определим *множество* U_X *вхождений символа* X *в грамматику* G. Это множество состоит из всех пар

(правило $p \in \mathsf{P}$, номер i символа в правиле p)

таких, что символ, стоящий на i-ом месте в правой части правила p является грамматическим символом X. Пару $(p,i) \in \mathsf{U}_X$ будем называть *вхождением символа* X *в правило* p.

Пусть t – токен. Для каждого вхождения $u \in \mathsf{U}_t$, $u = (p,i)$, $p = X \to \alpha t \beta$ токена t в грамматику G можно построить множество F_u токенов $t' \in \mathsf{T}$ таких, что существует вывод

$$S \overset{*}{\Rightarrow} \gamma X \delta \overset{p}{\Rightarrow} \gamma \alpha t \beta \delta \overset{*}{\Rightarrow} \alpha' t t' \beta'.$$

Здесь греческие буквы обозначают некоторые субсентенциальные формы, т.е. последовательности нетерминалов и токенов. Если в грамматике G существует вывод $S \overset{*}{\Rightarrow} \gamma X \overset{p}{\Rightarrow} \gamma \alpha t$ предложения, оканчивающегося токеном t, то будем считать, что множество F_u содержит пустую последовательность $\varepsilon \in \mathsf{F}_u$.

Через F_t будем обозначать объединение множеств F_u для токена t:

$$\mathcal{F}_t = \bigcup_{u \in U_t} \mathcal{F}_u.$$

Иными словами, множество F_t – это множество токенов, каждый из которых допустим для токена t в качестве следующего.

В дальнейшем нас главным образом будет интересовать дополнение к множеству F_t в множестве $\mathsf{T} \cup \{\varepsilon\}$. Будем обозначать это дополнение через

$$\mathfrak{N}_t = (\mathcal{T} \cup \{\varepsilon\}) \setminus \mathcal{F}_t.$$

Теорема 1. Последовательность токенов, содержащая подпоследователь-ность tt', где $t' \in \mathfrak{N}_t$, не является предложением языка, описываемого грамматикой G.

Доказательство. Очевидно из построения множества $\mathfrak{N}_t$. ►

Для последовательности токенов $\alpha = t_1 \ldots t_n$ такой, что существует вывод $S \overset{*}{\Rightarrow} \beta \alpha \gamma$, можно определить множество токенов $\mathfrak{N}_\alpha$такое, что, если $t' \in \mathfrak{N}_\alpha$, то не существует вывода $S \overset{*}{\Rightarrow} \beta \alpha t' \gamma$. Тогда любая последовательность $\beta \alpha t' \gamma$, где $t' \in \mathfrak{N}_\alpha$, не является предложением языка, описываемого грамматикой G.

Итак, мы научились получать последовательности токенов, заведомо не являющиеся предложениями целевого языка. К вопросу о произвольности негативной последовательности токенов мы вернемся в следующем параграфе.

3.2. Критерии покрытия

Как видно из описания LL- и LR-анализаторов, основной момент их работы – принятие решения о дальнейших действиях на основании некоторых неполных данных (прочитанной части входного потока). Для LL-анализатора ситуации выбора соответствует пара (нетерминал на вершине стека, текущий входной символ), а для LR-анализатора – пара (символ состояния конечного автомата на вершине стека, текущий входной символ). Отсюда возникают следующие критерии покрытия для позитивных тестовых наборов:

(PLL) Покрытие всех пар

(нетерминал A, допустимый следующий токен t),

где пара (A,t) считается покрытой тогда и только тогда, когда в тестовом наборе существует последовательность токенов, являющаяся предложением целевого языка, имеющая вывод $S \overset{*}{\Rightarrow} \alpha A \beta \Rightarrow \alpha t \gamma \beta$. Иными словами, LL-анализатор, обрабатывая эту последовательность, получит ситуацию, когда на вершине стека будет находиться символ A, а текущим входным символом будет токен t. Модификация этого критерия для расширенной формы BNF грамматики была сформулирована в работе [1].

(PLR) Покрытие всех пар

(символ s_i состояния конечного автомата,
помеченный символом X переход из состояния s_i),

где пара (s_i,X) считается покрытой тогда и только тогда, когда в тестовом наборе существует предложение языка, имеющее вывод $S \overset{*}{\Rightarrow} \alpha X \beta$ такой, что префикс α отвечает состоянию s_i. Или, что то же самое, LR-анализатор, обрабатывая это предложение получит ситуацию, когда на вершине стека будет находиться символ s_i, а началом текущего входного потока будет последовательность токенов, отвечающая символу X.

Аналогично возникают следующие критерии покрытия и для негативных тестовых наборов (эти критерии имеют параметр r – количество “правильных” токенов, предшествующих “неправильному” токену):

($\mathrm{NLL_R}$) Пусть A – нетерминал. Последовательность токенов $t_1 \ldots t_r$ назовем *допустимой для* A *предпоследовательностью токенов*, если существует сентенциальная форма $\alpha t_1 \ldots t_r A \beta$, выводимая из стартового правила. Рассмотрим объединение множеств $\mathfrak{N}_{t_1 \ldots t_r}$ по всем допустимым для A предпоследовательностям токенов длины $r \leq R$. Критерий состоит в том, что все пары (A,t'), где t' из рассмотренного объединения, должны быть покрыты. Здесь покрытие пары (A,t') означает, что среди тестов имеется последовательность токенов, не принадлежащая целевому языку, такая, что LL-анализатор, обрабатывая эту последовательность,

получит ситуацию, когда на вершине стека будет находиться символ A, а текущим входным символом будет "некорректный" символ t'.

(NLR_R) Пусть s_i – символ состояния конечного автомата, определяющего активные префиксы. Последовательность токенов $t_1...t_r$ назовем *допустимой для s_i предпоследовательностью токенов*, если существует выводимая из стартового правила последовательность токенов $\alpha t_1...t_r\beta$ такая, что ее префикс $\alpha t_1...t_r$ отвечает состоянию s_i. Рассмотрим объединение множеств $\mathfrak{N}_{t_1...t_r}$ по всем допустимым для s_i предпоследовательностям токенов длины $r \le R$. Критерий состоит в том, что все пары (s_i,t'), где t' из рассмотренного объединения, должны быть покрыты. Здесь покрытие пары (s_i,t') означает, что среди тестов имеется последовательность токенов, не принадлежащая целевому языку, такая, что LR-анализатор, обрабатывая эту последовательность получит ситуацию, когда на вершине стека будет находиться символ s_i, а текущим входным символом будет t'.

Для получения ситуации (A,t') в критерии (*NLL*) или ситуации (s_i,t') в критерии (*NLR*) требуется, чтобы парсер нормально проработал какое-то время, а затем встретил неверный символ. Для достижения этой цели необходимо, чтобы токены, идущие в последовательности до неверного токена t', образовывали префикс некоторого предложения языка, при разборе которого возникала бы требуемая ситуация в стеке. Поэтому в качестве негативного теста мы будем рассматривать измененное (с помощью вставки или замены токенов) предложение целевого языка так, чтобы в нем содержалась неправильная последовательность токенов $t_1...t_rt'$, где $t'\in \mathfrak{N}_{t_1...t_r}$.[2]

Завершая этот параграф, введем еще два полезных критерия покрытия для грамматик специального вида.

Пусть грамматика G такова, что ее каноническая система множеств пунктов удовлетворяет следующему свойству: если I_i и I_j — два различных множества из канонической системы, то множества базисных пунктов из *Ii* и *Ij* не пересекаются. Заметим, что для такой грамматики покрытие всех пар (состояние конечного автомата, переход из этого состояния в другое)

[2] Существует связь между предлагаемым подходом и методом мутационного тестирования. Именно, для любой последовательности токенов, являющейся негативным тестом в описанном выше смысле (т.е. предложением языка, "испорченным" с помощью вставки/замены "нехорошего" токена) можно построить грамматику-мутант такую, что данный негативный тест будет являться предложением языка, описываемого этой грамматикой-мутантом.

Одним из принципов мутационного тестирования является так называемый эффект взаимосвязи (coupling effect): при обнаружении простых ошибок будут обнаруживаться и более сложные (см. [16]). Согласно этому эффекту взаимосвязи, мутации должны быть простыми. Заметим, что предлагаемый подход вполне согласуется с этим принципом.

достигается при покрытии всех пунктов грамматики. Рассмотрим следующий критерий покрытия для наборов позитивных тестов:

(WPLR) Покрытие всех пар (пункт $\pi = B \to \alpha\bullet X\beta$ грамматики G, допустимый первый токен t для символа X). Пара (π,t) считается покрытой тогда и только тогда, когда в тестовом наборе существует предложение языка, имеющее вывод $S\overset{*}{\Rightarrow}\gamma B\delta\overset{*}{\Rightarrow}\gamma\alpha X\beta\delta\overset{*}{\Rightarrow}\gamma\alpha t\mu\beta\delta$.

Для грамматик указанного типа этот критерий является более сильным, чем критерий *PLR*. Действительно, нетрудно показать, что каждое состояние определяется множеством своих базисных пунктов. Отсюда, поскольку для грамматик указанного класса подмножества базисных пунктов у разных состояний не пересекаются, то покрыв все пункты, мы покроем и все состояния.

Аналогично можно сформулировать критерий покрытия для наборов негативных тестов:

($WNLR_R$) Пусть $\pi = B \to \alpha\bullet\beta$ – пункт грамматики G. Последовательность токенов $t_1...t_r$ назовем *предпоследовательностью токенов допустимой для π*, если существует выводимая из стартового правила последовательность токенов $\mu t_1...t_r\bullet\lambda$, имеющая вывод

$$S \overset{*}{\Rightarrow} \gamma B\delta \overset{*}{\Rightarrow} \gamma\alpha\beta\delta \overset{*}{\Rightarrow} \mu t_1 \dots t_r\bullet\lambda,$$

т.е. $\mu t_1...t_r$ выводится из $\gamma\alpha$, а λ – из $\beta\delta$. Рассмотрим объединение множеств $\mathfrak{N}_{t_1...t_r}$ по всем допустимым для π предпоследовательностям токенов длины $r \le R$. Критерий состоит в том, что все пары (π,t'), где t' из рассмотренного объединения, должны быть покрыты. Здесь пок-рытие пары (π,t') означает, что среди тестов имеется последовате-льность токенов, не принадлежащая целевому языку, такая, что некоторый ее префикс имеет вид $\mu t_1...t_rt'$, где $t_1...t_r$ – некоторая допус-тимая для π предпоследовательность токенов такая, что $t'\in \mathfrak{N}_{t_1...t_r}$.

3.3. Тестовые наборы

В данном разделе мы опишем способы генерации тестовых наборов, удовлетворяющих некоторым из описанных выше критериев покрытия.

3.3.1. Позитивные тесты

Пусть B – нетерминал, A – произвольный грамматический символ. Множество всех вхождений (p,i) символа A в грамматику, таких что $p = B \to \alpha A\beta$, обозначим через $\bigcup_A^B$. Упорядочим и перенумеруем элементы множества $\bigcup_A^B$, j-й элемент этого множества обозначим через $A^{(B,j)}$.

Пример. Если нетерминал B определяется тремя правилами

$$B \to ADE$$
$$B \to CADA$$
$$B \to KB,$$

то для символа A имеем три вхождения: $A^{(B,1)}$ из первого правила ($B \to A_1DE$), $A^{(B,2)}$ и $A^{(B,3)}$ из второго правила ($B \to CA_2DA_3$). В третьем правиле вхождений символа A нет. ►

Введем следующее обозначение: будем писать $A \prec B$, если A входит в правую часть какого-либо правила, определяющего нетерминал B.

Теорема 2. Для любого грамматического символа A, выводимого из стартового символа, существует цепочка $A \prec B_1 \prec \ldots \prec B_k \prec S$, где S – стартовый символ грамматики, а B_i – нетерминалы, такая, что все нетерминалы, входящие в эту цепочку, различны.

Доказательство. То, что символ A выводим из стартового символа, означает, что для символа A существует какая-то цепочка $A \prec B_1 \prec \ldots \prec B_k \prec S$. Пусть это минимальная по длине такая цепочка. Пусть теперь в цепочке $B_i \prec \ldots \prec B_k \prec S$ нет повторяющихся нетерминалов, а в цепочке $B_{i\text{-}1} \prec \ldots \prec B_k \prec S$ – есть, т.е. $B_{i\text{-}1} = B_l$ для некоторого $l = i,\ldots,k$. Но тогда мы можем заменить цепочку $A \prec B_1 \prec \ldots \prec B_k \prec S$ на цепочку $A \prec B_1 \prec \ldots \prec B_{i\text{-}2} \prec B_l \prec \ldots \prec B_k \prec S$, меньшую по длине, что противоречит минимальности первоначальной цепочки. ►

Доказанная теорема показывает, как можно строить наборы тестов, удовлетворяющие критерию (*PLL*). Действительно, эта теорема дает следующий способ построения цепочки $A \prec B_1 \prec \ldots \prec B_k \prec S$:

- Для символа A строим множество N_A нетерминалов, для которых существует определяющее правило, содержащее A в правой части. Если $S \in N_A$, то искомая цепочка построена.
- Для каждого элемента B_1 множества N_A строим множества N_{A,B_1} нетерминалов $B_2 \neq B_1$, для которых существует определяющее правило, содержащее B_1 в правой части. Если $S \in N_{A,B_1}$, то искомая цепочка построена.

  ...
- Для каждого элемента $B_{s\text{-}1}$ множества $N_{A,B_1,\ldots,B_{S-2}}$ строим множества $N_{A,B_1,\ldots,B_{S-1}}$ нетерминалов $B_s \notin \{B_1,\ldots,B_{s\text{-}1}\}$, для которых существует определяющее правило, содержащее $B_{s\text{-}1}$ в правой части. Если $S \in N_{A,B_1,\ldots,B_{S-2}}$, то искомая цепочка построена.

  ...

Из теоремы 2 следует, что приведенный алгоритм завершит работу построением искомой цепочки. Заметим теперь, что, имея цепочку $A \prec B_1 \prec \ldots \prec B_k \prec S$, можно получить из нее несколько выводов сентенциальных форм, если конкретизировать соответствующие вхождения. Каждая из этих сентенциальных форм будет иметь вид $\alpha A\beta$, где α, β – некоторые последовательности грамматических символов.

Определим функцию *first*(A), возвращающую множество раскрытий символа A:

- Если A – токен, то *first*(A) = $\{A\}$.
- Если A – нетерминал, то
$$first(A) = \bigcup_{p=A\to\alpha} first(\alpha).$$
- Если $\alpha = A_1 \ldots A_n$ – сентенциальная форма, то пусть
$$C_A = \{\beta A_2 \ldots A_n \mid \beta \in first(A_1)\}.$$
Тогда, если A_1 – грамматический символ, не имеющий пустого раскрытия, то *first*(α) = C_A, а если A_1 имеет пустое раскрытие, то *first*(α) = $C_A \cup$ *first*($A_2 \ldots A_n$).

При этом рекурсия при построении *first*(A) обрывается так: если при построении *first*(A) необходимо это же множество, то его считают пустым.

Теорема 3. Множество *first*(A) включает раскрытия A со всевозможными первыми токенами для A.

Доказательство. Следует из построения. ►

Имея выведенную из стартового символа сентенциальную форму $\alpha A\beta$ и множество *first*(A), мы можем построить множество сентенциальных форм $\alpha\gamma\beta$, где $\gamma \in$ *first*(A). Для каждой такой сентенциальной формы выберем какую-нибудь последовательность токенов, выводимую из этой сентенциальной формы. Множество выбранных последовательностей обозначим через $\mathfrak{T}_A$.

Теорема 4. Объединение множеств $\mathfrak{T}_A$ для всех нетерминалов A является множеством позитивных тестов, удовлетворяющим критерию (*PLL*).

Доказательство. Следует из построения и из теоремы 3. ►

Здесь же можно указать способ построения набора позитивных тестов, удовлетворяющего критериям *WPLR* и *PLR*.

Пусть B – нетерминал из грамматики G, обозначим через $\mathsf{D}_k(B)$ множество сентенциальных форм вида $\alpha B\beta$, выводимых из стартового правила, в цепочке вывода которых каждое правило встречается не более чем k раз.

Пусть $\pi = B \to \lambda \bullet X\mu$ – пункт грамматики G. Положим
$$first(\pi) = \{\lambda\gamma\mu \mid \gamma \in first(X)\}.$$

Пусть $\zeta = \alpha B \beta \in \mathsf{D}_k(B)$. Для каждой сентенциальной формы $\alpha\lambda\gamma\mu\beta$, где $\lambda\gamma\mu \in first(\pi)$, выберем какую-нибудь последовательность токенов, выводимую из этой сентенциальной формы. Множество выбранных последовательностей обозначим через $\mathfrak{T}_{\pi,\zeta}$. Очевидно, что всякая последовательность токенов из $\mathfrak{T}_{\pi,\zeta}$ является предложением целевого языка, а значит – позитивным тестом для парсера.

Теорема 5. Пусть для каждого нетерминала $B \in \mathsf{N}$ грамматики G фиксирована сентенциальная форма $\zeta_B \in \mathsf{D}_k(B)$. Тогда множество позитивных тестов:

$$\bigcup_{\pi \in \mathfrak{P},\, \pi = B \to \ldots} \mathfrak{T}_{\pi,\zeta_B}$$

удовлетворяет критерию (*WPLR*); здесь $\mathfrak{P}$, как и раньше, обозначает множество всех пунктов грамматики G.

Доказательство. Следует из теоремы 3. ►

Теорема 6. Начиная с некоторого k, множество позитивных тестов

$$\bigcup_{\pi \in \mathfrak{P}} \bigcup_{\zeta \in \mathsf{D}_k(B)} \mathfrak{T}_{\pi,\zeta}$$

удовлетворяет критерию (*PLR*).

Доказательство. Поскольку при стремлении k к бесконечности множество $\mathsf{D}_k(B)$ стремится к множеству всех сентенциальных форм вида $\alpha B \beta$, то, начиная с некоторого k, все возможные состояния автомата, определяющего активные префиксы, будут покрыты. Так как мы берем объединение по всем пунктам, то будут покрыты и все пары (s_i, X), где s_i – символ состояния конечного автомата, X – символ грамматики. ►

3.3.2. Негативные тесты

Ранее мы уже заметили, что наиболее естественный способ получения негативного теста – мутация некоторого предложения целевого языка, т.е. замена в этом предложении некоторого участка (возможно пустого) на “неправильную” последовательность токенов.

Пусть α – предложение языка, и пусть в нем отмечен токен с порядковым номером i. *Мутацией 1-го рода* (обозначается $mut_1(\alpha,i)$) мы будем называть замену предложения $\alpha = t_1 \ldots t_n$ на множество последовательностей токенов вида $t_1 \ldots t_i t' t_{i+1} \ldots t_n$, где $t' \in \mathfrak{N}_{t_i}$. При мутации 1-го рода в предложение *вставляются* “неправильные” токены. *Мутацией 2-го рода* (обозначается $mut_2(\alpha,i)$) будем называть замену предложения $\alpha = t_1 \ldots t_n$ на множество последовательностей токенов вида $t_1 \ldots t_i t' t_{i+2} \ldots t_n$, где $t' \in \mathfrak{N}_{t_i}$. При мутации 2-го рода в предложении один из токенов *заменяется* на “неправильный” токен. Кроме того, можно определить мутации 1-го и 2-го рода при $i = 0$. В этом случае t' принадлежит множеству недопустимых первых символов, т.е.

$$t' \in \mathcal{T} \setminus \{t \mid \exists \beta = x_1 \ldots x_m \in \mathfrak{L}_G,\, x_1 = t\},$$

а операции мутации определяются как и ранее: $mut_1(t_1 \ldots t_n, 0) = \{t' t_1 \ldots t_n\}$ и $mut_2(t_1 \ldots t_n, 0) = \{t' t_2 \ldots t_n\}$.

Из теоремы 1 следует, что обе операции мутации дают негативные тесты. Перейдем к описанию способа получения множества предложений с отмеченными токенами, позволяющего строить множества тестов, удовлетворяющие критериям (NLL_1) и (NLR_1).

Пусть $\pi = D \to B_1 \ldots B_i \bullet B_{i+1} \ldots B_n$ – некоторый пункт грамматики G.

Рассмотрим следующий алгоритм построения множества предложений с отмеченными токенами.

- Пусть M_π – пустое множество.
- Построим сентенциальную форму $\alpha D \beta \overset{*}{\Leftarrow} S$, выводимую из стартового символа (как это сделать, обсуждалось выше). Пусть γ – непустая последовательность из множества $first(B_{i+1} \ldots B_n)$. Добавим в M_π все сентенциальные формы вида $\alpha\gamma\beta$, в которых отмечен первый символ последовательности γ. Нетрудно видеть, что этот символ является токеном (это следует из построения множества $first(\ldots)$). Если последовательность $B_{i+1} \ldots B_n$ может быть раскрыта в пустую последовательность токенов, то делаем шаг 3, иначе M_π вычислено.
- Для каждого пункта π' такого, что $\pi' = X \to \lambda D \bullet \mu$, добавим в M_π все сентенциальные формы из множества $M_{\pi'}$. При этом рекурсия при построении M_π обрывается так: если при построении M_π необходимо само это множество, его считают пустым.

Рассмотрим грамматику $\mathfrak{G}$, полученную из грамматики G следующим образом. Множества токенов и нетерминалов, а также стартовый символ в грамматиках $\mathfrak{G}$ и G совпадают. Если в грамматике G присутствует правило вывода $A \to B_1 \ldots B_n$, то в грамматике $\mathfrak{G}$ присутствует правило $A \to B_n \ldots B_1$, здесь A – нетерминал, а B_i – грамматические символы. Других правил в грамматике $\mathfrak{G}$ нет.

Очевидно, что грамматика $\mathfrak{G}$ задает язык, каждое предложение которого – это некоторое предложение языка, задаваемого грамматикой G, написанное “задом наперед”.

Возьмем пункт $\bar{\pi} = D \to B_n \ldots B_{i+1} \bullet B_i \ldots B_1$ в грамматике $\mathfrak{G}$, соответствующий пункту π в грамматике G. Вычислим $M_{\bar{\pi}}$ в грамматике $\mathfrak{G}$. Пусть $N_\pi = \{(X_1 \ldots X_n, i) \mid (X_n \ldots X_1, i) \in M_{\bar{\pi}}\}$. Каждую сентенциальную форму из множества N_π можно раскрыть в одно или несколько предложений целевого языка с отмеченным токеном, если как-нибудь раскрыть все входящие в эту сентенциальную форму нетерминалы. Обозначим полученное множество помеченных предложений через $\mathfrak{S}_\pi$. К нему можно применить ранее определенные операции мутации 1-го и 2-го рода.

Теорема 7. Пусть A – нетерминал и $\mathfrak{P}_A$ – множество пунктов $D \to B_1...B_i \bullet B_{i+1}...B_n$ таких, что $B_{i+1} = A$. Тогда множества негативных тестов

$$\mathfrak{S}_t = \bigcup_A \bigcup_{\pi \in \mathfrak{P}_A} mut_t(\mathfrak{S}_\pi), \quad t = 1,2$$

удовлетворяют критерию (NLL_1).

Доказательство. Действительно, пусть существует вывод $S \overset{*}{\Rightarrow} \alpha t A \beta$.

Нетерминал A получается на некотором шаге этого вывода, т.е.

$$S \overset{*}{\Rightarrow} \gamma D \delta \overset{R}{\Rightarrow} \gamma B_1 \dots B_i A B_{i+2} \dots B_n \delta \overset{*}{\Rightarrow} \alpha t A \beta,$$

где $p = D \to B_1...B_iAB_{i+2}...B_n$. По условию, для пункта $\bar{\pi} = D \to B_n...B_{i+2}A \bullet B_i...B_1$ из грамматики $\mathfrak{G}$ было построено множество $M_{\bar{\pi}}$. Отсюда ясно, что, если в $M_{\bar{\pi}}$ существует последовательность вида $\lambda A t \mu$, то ситуация (A,t'), где $t' \in \mathfrak{N}_t$, покрыта.

Заметим, что исходный вывод предложения $\alpha t A \beta$ может быть преобразован в вывод в грамматике $\mathfrak{G}$:

$$S \overset{*}{\Rightarrow} \delta' D \delta' \overset{R}{\Rightarrow} \delta' B_n \dots B_{i+2} A B_i \dots B_1 \gamma' \overset{*}{\Rightarrow} \beta' A t \alpha'.$$

Если последовательность $B_i...B_1$ не имеет пустого раскрытия, то существует такая последовательность из $first(B_i...B_1)$, что t является ее первым токеном, т.е. в множестве $M_{\bar{\pi}}$ имеется сентенциальная форма $\lambda A t \mu$, что и требовалось. Случай, когда $B_i...B_1$ может быть раскрыта в пустую последовательность, разбирается аналогично. ►

Теорема 8. Множества негативных тестов

$$\mathfrak{S}_t = \bigcup_A \bigcup_{\pi \in \mathfrak{P}_A} mut_t(\mathfrak{S}_\pi), \quad t = 1,2$$

определенные в теореме 7, удовлетворяют критерию ($WPLR_1$).

Доказательство. Аналогично доказательству теоремы 7. ►

Пусть $\pi = D \to B_1...B_i \bullet B_{i+1}...B_n$ – пункт грамматики G. Заметим, что на втором этапе построения M_π мы выбирали сентенциальную форму $S \overset{*}{\Rightarrow} \alpha D \beta$, так что в действительности построенное M_π, а значит и $\mathfrak{S}_\pi$, зависят от этой сентенциальной формы. Пусть $\mathsf{D}_k(D)$ – множество сентенциальных форм вида $\alpha D \beta$, в цепочке вывода которых каждое правило встречается не более k раз.

Теорема 9. Начиная с некоторого k, множества негативных тестов

$$\mathfrak{S}_{tk} = \bigcup_{\pi \in \mathfrak{P}} \bigcup_{\alpha D \beta \in \mathsf{D}_k} mut_t(\mathfrak{S}_\pi), \quad t = 1,2$$

удовлетворяют критерию (NLR_1).

Доказательство. Поскольку при стремлении k к бесконечности множество $\mathsf{D}_k(D)$ стремится к множеству всех сентенциальных форм вида $\alpha D \beta$, то, начиная с некоторого k, все возможные состояния автомата, определяющего активные

префиксы, будут покрыты. Таким образом, будут покрыты и все пары (s_i,t'), где s_i – символ состояния конечного автомата, t' – негативный токен. ►

4. Практические результаты.

Для апробации предложенной методики нами были построены следующие генераторы:

- Генератор G_P множества позитивных тестов для парсера, удовлетворяющий критерию PLL.
- Генератор G_N множества негативных тестов для парсера, удовлетворяющий критериям $WNLR_1$ и NLL_1.

Оба генератора получают на вход BNF целевого языка. У генератора G_P имеются параметры, позволяющие регулировать количество тестов, а у генератора G_N – параметр, позволяющий указывать способ мутации.

Кроме того, генератор G_N генерирует документированные тесты, т.е. в каждом тесте имеется комментарий о том, в каком месте следует ожидать ошибку и какого рода эта ошибка. Таким образом, если парсер имеет дополнительную функциональность – выдачу сообщения об ошибке при ее обнаружении, то можно протестировать и эту функциональность, если сравнить ожидаемые и полученные данные об ошибке.

Генератором G_P были сгенерированы наборы тестов для BNF следующих языков:

- C – расширение ANSI C, для которого реализован известный компилятор gcc (см. [19]);
- mpC – расширение ANSI C, разработанное для программирования параллельных вычислений (см. [23]);
- Java (версия 1.4);
- J@va – спецификационное расширение языка Java, разработанное в ИСП РАН (см. [21]).

Некоторые данные об использовавшихся грамматиках приведены в таблице:

Характеристики	**C**	**Java**	**J@va**
Количество нетерминалов	70	135	174
Количество токенов	93	101	140
Минимальная мощность $\mathfrak{N}_t$	1	19	34
Максимальная мощность $\mathfrak{N}_t$	93	101	140
Средняя мощность $\mathfrak{N}_t$	68	79	119

Генератором G_N были сгенерированы наборы тестов для BNF трех указанных языков (исключая mpC). Количественные характеристики сгенерированных тестовых наборов приводятся в таблицах:

Генератор	Количество тестов		
	C	**Java**	**J@va**
G_P	137307	137148	219221
G_N	43448	71943	145758

Генератор	Время генерации		
	C	**Java**	**J@va**
G_P	19 m 30 s	19 m 51 s	43 m 08 s
G_N	3 m 01 s	7 m 14 s	19 m 25 s

Генератор	Средний размер теста в байтах		
	C	**Java**	**J@va**
G_P	58	68	112
G_N	219	220	266

Позитивные тесты для Java и J@va были прогнаны через парсер языка J@va, разработанный в ИСП РАН (см. [21]). Указанный парсер был разработан при помощи JavaCC (см. [20]). В результате прогона тестов было обнаружено восемь ошибок. Позитивные тесты для языка mpC обнаружили 12 ошибок в компиляторе этого языка. Позитивные тесты для C были прогнаны на широко используемом компиляторе gcc (см. [19]). При этом на 112 тестах gcc зациклился. Негативные тесты для C были прогнаны на парсере языка C, разрабатываемом в ИСП РАН (см. [22]), и обнаружили две ошибки.

5. Заключение

На основе модельного подхода к тестированию были разработаны критерии покрытия для наборов позитивных и негативных тестов для парсеров.

Одно из достоинств предложенного в статье метода построения негативных тестов для синтаксического анализатора – гарантия негативности построенных тестов. Это позволяет отказаться от использования каких бы то ни было эталонных синтаксических анализаторов.

Были разработаны методы генерации наборов тестов, удовлетворяющих предложенным критериям. С помощью этих методов были построены генераторы позитивных и негативных тестовых наборов. Сгенерированные для разных языков тестовые наборы были использованы для тестирования ряда парсеров и компиляторов и выявили в тестируемых компонентах ошибки.

Таким образом, практические результаты применения данной методики показывают адекватность предложенных критериев покрытия, а также целесообразность использования предложенной методики для тестирования промышленных компиляторов.

Литература

1. А. В. Демаков, С. В. Зеленов, С. А. Зеленова. *Тестирование парсеров текстов на формальных языках* // Программные системы и инструменты (тематический сборник факультета ВМиК МГУ). – 2001. – № 2. – p. 150-156.
2. С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. *Генерация тестов для компиляторов и других текстовых процессоров* // Программирование, Москва. – 2003. – 29. – № 2. – с. 59-69.
3. С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. *Применение модельного подхода для автоматического тестирования оптимизирующих компиляторов* // http://www.citforum.ru/SE/testing/compilers/
4. А.К. Петренко и др. *Тестирование компиляторов на основе формальной модели языка* // Препринт института прикладной математики им. М.В Келдыша. 1992. №45.
5. A .Aho, R. Sethi, J. D. Ullman. *Compilers. Principles, Techniques, and Tools* // Addison-Wesley Publishing Company, Inc. – 1985.
6. R. A. DeMillo, A. J. Offut. *Constraint-Based Automatic Test Data Generation* // IEEE Transactions on Software Engineering. – 1991. – 17. – № 9. – p. 900-910.
7. R. F. Guilmette. *TGGS: A flexible system for generating efficient test case generators.* 1999.
8. J. Harm, R. Lämmel. *Two-dimensional Approximation Coverage* // Informatica Journal. – 2000. – 24. – № 3.
9. R. Lämmel. *Grammar testing* // In Proc. of Fundamental Approaches Software Engineering. – 2001. – 2029. – p. 201-216.
10. R. Lämmel, C. Verhoef. *Cracking the 500-Language Problem* // IEEE Software. – 2001. – 18. – № 6. – p. 78-88.
11. P. M. Maurer. *Generating test data with enhanced context-free grammars* // IEEE Software. – 1990. – p. 50-55.
12. P. M. Maurer. *The design and implementation of a grammar-based data generator* // Software Practice and Experience. – 1992. – 22(3). – p. 223-244.
13. W. McKeeman. *Differential testing for software.* // Digital Technical Journal. – 1998. – 10(1). – p. 100-107.
14. M. Mernik, G. Gerlic, V. Zumer, B. R. Bryant. *Can a parser be generated from examples?* // Symposium on Applied Computing, Proceedings of the 2003 ACM symposium on Applied computing, Session: Programming languages and object technologies, ISBN:1-58113-624-2 – 2003. – p. 1063-1067.
15. A. J. Offut, S. D. Lee. *An Empirical Evaluation of Weak Mutation* // IEEE Transactions on Software Engineering. – 1994. – 20. – № 5. – p. 337-344.
16. A. J. Offut, R. H. Untch. *Mutation 2000: Uniting the Orthogonal* // Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA. – October, 2000. – p. 45-55.
17. A. K. Petrenko. *Specification Based Testing: Towards Practice* // LNCS. –2001. – 2244. – p. 287-300.
18. P. Purdom. *A Sentence Generator For Testing Parsers* // BIT. 1972. – № 2. – p. 336-375.
19. GCC. http://gcc.gnu.org/
20. JavaCC. https://javacc.dev.java.net/
21. J@T. http://unitesk.com/products/jat/
22. CTesK. http://unitesk.com/products/ctesk/
23. ”The mpC Programming Language Specification”, The Institute for System Programming of Russian Academy of Science. http://www.ispras.ru/~mpc.