

25 января 2009 г.

П. В. Егоров

РАСШИРЕНИЕ СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ С СОХРАНЕНИЕМ ОБРАТНОЙ аСОВМЕСТИМОСТИ

1. Введение

Все разработчики языков программирования (ЯП) стоят перед выбором: сделать язык максимально богатым, включив в него поддержку многих парадигм и технологий программирования, либо ограничиться небольшим ядром языка, поместив все остальное в различные библиотеки. Типичными примерами первого подхода являются языки Perl и C++. Примерами второго подхода – языки семейства Pascal. У каждого из этих подходов есть как достоинства, так и недостатки.

В данной работе рассматривается способ решения этого противоречия – расширяемый синтаксис языка программирования. Основная идея языка программирования с расширяемым синтаксисом заключается в том, чтобы с одной стороны ограничиваться небольшим подмножеством языка везде, где этого достаточно, а в модулях, требующих интенсивной работы с некоторой технологией или парадигмой, подключать соответствующее расширение синтаксиса. Таким образом, язык можно сделать максимально мощным, но при этом позволить программисту полностью контролировать богатство языка в каждый момент времени. Эта идея была озвучена довольно давно.

Существуют впечатляющие примеры успешных, гибко расширяющихся программных продуктов, например, интернет пейджер Miranda IM, или среда разработки Eclipse. Богатство функциональных возможностей в них сочетаются с гибкостью добавления новых возможностей. Разрабатывая и подключая к ядру программы дополнительные модули можно расширять возможности программы. Благодаря опубликованному описанию интерфейса создавать дополнительные модули могут сторонние разработчики и вообще любые желающие. Аналогичные идеи можно попробовать применить и к устройству языков программирования.

Для того чтобы лучше проиллюстрировать выгоды, которые получают программисты от такого подхода к созданию трансляторов, рассмотрим пример использования гипотетического языка с расширяемым синтаксисом.

Пример 1.1. *Использование ЯП с расширяемым синтаксисом.*

```
syntax Sql;  
  
string name = "Иван";  
SqlQuery q = sql(select count(*) from Persons where name=$name);  
int count = connection.Execute(q);
```

Сначала командой `syntax Sql` подключается расширение синтаксиса, добавляющее поддержку технологии sql-запросов. Это позволяет писать sql запросы прямо в исходном коде. Благодаря этому среда разработки может подсвечивать синтаксис sql запроса, а компилятор проверять его корректность непосредственно на этапе компиляции. Основные выгоды от использования ЯП с расширяемым синтаксисом можно сформулировать следующим образом:

1. Для решения каждого класса специфических задач можно создать расширение с максимально удобным синтаксисом.
2. Многие проверки переходят с этапа выполнения на этап компиляции, увеличивая скорость обнаружения ошибок. В данном примере так случилось с проверкой корректности sql-запроса.

Важно подчеркнуть, что различные расширения ЯП могут создаваться группами разработчиков, независимыми от группы, создавшей ядро языка. Этот факт может привести к качественному увеличению скорости развития языков и парадигм программирования. Раньше, для проверки практикой какой-то новой концепции создавались целые новые языки. Так, например, дизайн по контракту был реализован в специально созданном для этого языке Eiffel. Проверяемые исключения впервые были опробованы в языке Java. Если бы можно было вводить поддержку своих идей в свой любимый язык программирования, новые концепции стали изобретаться бы и отбраковываться гораздо быстрее и эффективнее.

Итак, преимущества довольно внушительные, и это подтверждается многочисленными попытками создать язык с расширяемым синтаксисом. В качестве примеров таких языков можно назвать, например, язык Haskell с его метапрограммированием ([2]), и новые, развивающиеся языки Nemerle ([3, 4]) и Boo с их системой макросов. Сама же идея того, что будущее программной инженерии за гибкими, расширяемыми языками была озвучена довольно давно. Например, ее можно увидеть в статье Extensible Programming for the 21st Century (Расширяемое программирование для 21-ого века) [5].

В данной работе будет рассмотрен несколько более более низкоуровневый и гибкий по сравнению с макросами подход к решению задачи расширения синтаксиса.

Первым делом нужно определиться с термином «расширение». Пусть есть некоторый транслируемый язык L_0 . Для начала зафиксируем неформальное определение. Язык L_1 можно называть расширением L_0 , если он сохраняет все возможности расширяемого языка, добавляя какие-то новые возможности. Наилучшим вариантом является сохранение обратной совместимости расширения с исходным языком.

Будем называть язык L_1 и его транслятор T_1 **совместимым расширением** языка L_0 и его транслятора T_0 , если любая корректная программа на языке L_0 будет корректной и в языке L_1 , а кроме того, результат трансляции этой программы в языке L_1 останется тем же, что и в языке L_0 .

Обычно процесс трансляции разбивают на несколько последовательных этапов. Создание ЯП с расширяемым синтаксисом означает превращение каждого такого этапа в расширяемый. При этом контролировать сохранение обратной совместимости можно на каждом этапе по отдельности: если обратная совместимость будет сохраняться при расширении на каждом из этапов трансляции, то все расширение транслятора целиком также будет обладать обратной совместимостью с исходным языком. Сегодня выделение в качестве двух первых этапов трансляции лексического и синтаксического анализов стало стандартом де-факто. В этой работе будут рассмотрены подходы к созданию расширяемой версий синтаксического анализатора, сохраняющего совместимость при расширении.

Обозначим через CFG множество всех контекстно-свободных однозначных грамматик. Далее мы будем иметь дело только с такими грамматиками.

Ниже будут использоваться следующие общепринятые обозначения. ([1, стр. 27]) Нетерминальные символы грамматики будут обозначаться прописными латинскими буквами. Строчными латинскими буквами из конца алфавита будут обозначаться слова над алфавитом терминальных символов. Терминальные символы – строчными латинскими буквами из начала алфавита. Слова из терминальных и нетерминальных символов – греческими маленькими буквами.

2. Расширяющие преобразования грамматик

Этап синтаксического анализа управляется грамматикой соответствующего языка. Соответственно, расширение синтаксического анализатора может заключаться в расширении соответствующей грамматики.

Определение 2.1. Преобразование грамматик $\phi : CFG \rightarrow CFG$ называется расширяющим преобразованием грамматик, если

$$\forall G \in CFG \Rightarrow L(G) \subseteq L(\phi(G))$$

Далее будет дано определение множества AE расширяющих преобразований грамматик, удобных для использования и исследования. В этой работе мы ограничимся только преобразованиями из этого класса AE .

Определение 2.2. Определим множество преобразований Add как множество всех преобразований $\phi : CFG \rightarrow CFG$, добавляющих в грамматику новое правило. Другими словами таких, что выполняются следующие условия:

$$\begin{aligned}\phi(V, \Sigma, P, S) &= (V_1, \Sigma_1, P_1, S), \\ P &\subset P_1, |P_1 \setminus P| = 1.\end{aligned}$$

Преобразование из Add может увеличить как терминальный, так и нетерминальный алфавит. Кроме того, результат преобразования из Add может оказаться неприведенной грамматикой.

Некоторые авторы ([1, П.1, стр. 559], [6, 4.1, стр. 29]) при обсуждении расширения синтаксиса ограничиваются лишь этим классом расширяющих преобразований. В данной же работе предлагается ввести еще один класс преобразований, делающий расширение грамматики более гибким.

Определение 2.3. Определим множество преобразований $Extract$ как множество всех преобразований $\phi : CFG \rightarrow CFG$ таких, что выполняются следующие условия:

1. $\phi(V, \Sigma, P, S) = (V_1, \Sigma, P_1, S)$
2. $V_1 = V \cup \{B\}, B \notin V$
3. $P_1 = (P \setminus \{A \rightarrow \alpha\beta\gamma\}) \cup \{A \rightarrow \alpha B\gamma, B \rightarrow \beta\}$, для некоторого правила $A \rightarrow \alpha\beta\gamma$ из P .

Другими словами, после применения преобразования из $Extract$, одно некоторое правило $A \rightarrow \alpha\beta\gamma$ заменяется двумя правилами:

$$A \rightarrow \alpha B\gamma$$

$$B \rightarrow \beta$$

Преобразования из *Extract* расширяют нетерминальный алфавит, но не расширяют терминальный. Кроме того они не изменяют языка грамматики, не изменяют свойства однозначности и приведенности грамматики.

Определение 2.4. $AE = Add \cup Extract$

В множестве AE преобразования *Extract* играют роль подготовки грамматики к расширению языка, а преобразования *Add* собственно расширяют язык. Интуитивно понятно, что наличие преобразований *Extract* снижает требования к тщательности проектирования грамматики исходного языка.

Далее мы будем работать только с расширениями, полученными композицией конечного количества преобразований из AE :

Определение 2.5. Пусть Φ – класс некоторых расширяющих преобразований. Обозначим через Φ^* множество всевозможных композиций преобразований из Φ .

Довольно легко заметить, что элементы AE^* – это расширяющие преобразования.

3. Синтаксический анализатор

Определение 3.1. Пусть L – некоторый язык. Тогда синтаксическим анализатором языка L будем называть произвольную функцию, определенную на L .

Таким образом на данном этапе мы не накладываем никаких ограничений на то, что является результатом работы синтаксического анализатора.

Определение 3.2. Пусть w – слово языка $L(G)$, а s – синтаксический анализатор языка $L(G)$. Тогда $s(w)$ будем называть внутренним представлением слова w в грамматике G .

Далее дадим более формальное определение обратной совместимости для синтаксических анализаторов.

Определение 3.3. Пусть $G_1, G_2 \in CFG$, причем $L(G_1) \subseteq L(G_2)$. Пусть s_1, s_2 – синтаксические анализаторы языков $L(G_1)$ и $L(G_2)$. Тогда s_2 будем называть совместимым с s_1 , если $s_2|_{L(G_1)} = s_1$

4. Выбор внутреннего представления

В задаче трансляции синтаксический анализатор занимается не только распознаванием корректных программ, но и построением некоторого внутреннего представления программы в качестве результата своей работы. Этот раздел содержит ряд примеров, на основе которых далее будет построено удобное внутреннее представление и соответствующий алгоритм синтаксического анализа.

Для начала попробуем использовать в качестве внутреннего представления классическое дерево вывода. Для некоторого слова w языка $L(G)$ обозначим через $s(w)$ дерево вывода w в G .

Ниже приведено два примера, поясняющих, почему дерево вывода является не самым удачным внутренним представлением в рамках задачи расширения синтаксиса.

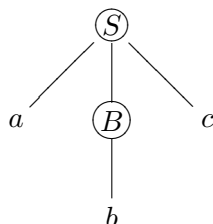
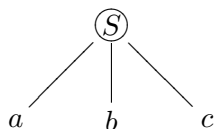
Пример 4.1. *Простейшее нарушение совместимости из-за цепных узлов, то есть узлов ровно с одним сыном.*

Рассмотрим две грамматики:

$$G_1 : S \rightarrow abc$$

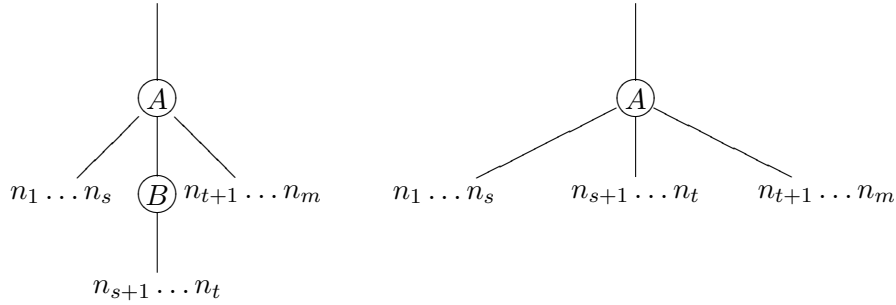
$$G_2 : S \rightarrow aBc; B \rightarrow b$$

Очевидно, G_2 получается из G_1 применением преобразования из *Extract*. Однако видно, что $s_1(abc) \neq s_2(abc)$.



Таким образом расширение грамматики с помощью преобразований из *AE* приводит к несовместимости исходного и расширенного синтаксических анализаторов. Значит такой выбор устройства синтаксических анализаторов не подходит для конструирования расширяемых трансляторов.

Вырезание цепных узлов, показанным ниже образом решает этот класс проблем.



Пример 4.2. *Нарушение совместимости из-за ε -листьев.*

Еще раз рассмотрим две грамматики:

$$G_1 : S \rightarrow a$$

$$G_2 : S \rightarrow aA; A \rightarrow \varepsilon$$

Аналогично G_2 получается из G_1 применением преобразования из *Extract*. Слово a содержится в обоих языках. А деревья вывода этого слова в G_1 и G_2 будут различными, даже если цепные узлы, раскрытые по правилу $A \rightarrow \varepsilon$ будут вырезаны из дерева вывода в G_2 .



Далее описано внутреннее представление, в основе которого лежит классическое дерево вывода, и устройство соответствующего синтаксического анализатора, лишенное этих двух недостатков.

5. Сокращенное дерево вывода

Пусть есть некоторая грамматика $G = (V, \Sigma, P, S)$ и p – это некоторое правило вывода из P . Через $len(p)$ будем обозначать длину правой части правила вывода p .

Определение 5.1. *Разметкой грамматики G будем называть функцию m , которая каждому правилу вывода грамматики G будет ставить в соответствие кортеж, состоящий из нулей и единиц, длиной в количество символов в правой части своего аргумента. Другими словами для любого правила p $m(p)$ есть кортеж*

$$(m_1, m_2, \dots, m_{len(p)}), \quad \text{где } m_i \in \{0, 1\}$$

Кортеж $m(p)$ будем называть разметкой правила p .

Для удобства введем следующее обозначение для элементов кортежа:

$$m(p) = (m(p, 1), m(p, 2), \dots, m(p, \text{len}(p))).$$

Значение $m(p, i)$ будем называть разметкой i -ого символа правой части правила p .

Будем обозначать разметку правила вывода с помощью верхнего индекса у символов правой части. Например для правила вывода $p : A \rightarrow BCde$ и разметки $m(p) = (0, 1, 0, 1)$, правило вывода будем обозначать так: $p : A \rightarrow B^0C^1d^0e^1$

Разметка нам потребуется для того, чтобы вырезать некоторые узлы из дерева вывода.

Определение 5.2. Определим множество размеченных грамматик Γ_M как множество всевозможных пар (G, m) , где $G \in CFG$, а m – разметка G . Саму пару (G, m) будем называть размеченной грамматикой.

Работая с размеченной грамматикой, каждому узлу кроме корня в классическом дереве вывода можно сопоставить значение разметки правила, в результате применения которого данный узел появился. Такое сопоставление будем называть разметкой узлов дерева вывода. Для единообразия доопределим разметку на корне дерева единицей. Чтобы пояснить идею разметки дерева вывода, обратимся к примеру:

Пусть дана следующая грамматика:

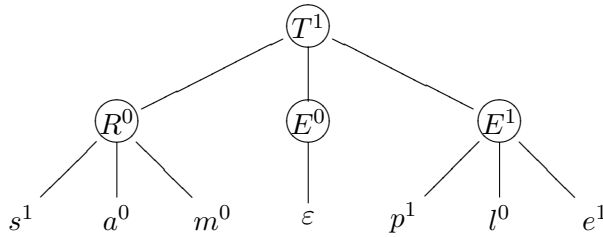
$$T \rightarrow R^0 E^0 E^1$$

$$R \rightarrow s^1 a^0 m^0$$

$$E \rightarrow \varepsilon$$

$$E \rightarrow p^1 l^0 e^1$$

Соответствующее дерево вывода слова «sample» в данной грамматике будет выглядеть так:



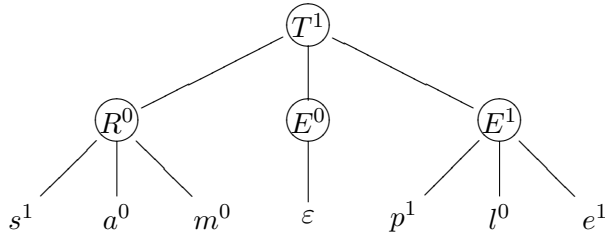
Определение 5.3. *Вспомогательным деревом вывода слова w в грамматике G , назовем дерево, полученное из обычного дерева вывода слова w в грамматике G , в результате удаления всех листьев, помеченных ε .*

Определение 5.4. *Сокращенным деревом вывода слова w в грамматике G , с разметкой t назовем дерево, полученное из вспомогательного дерева вывода слова w в грамматике G , путем вырезания из дерева всех узлов, размеченных нулями. Ниже дано определение вырезания узла из дерева.*

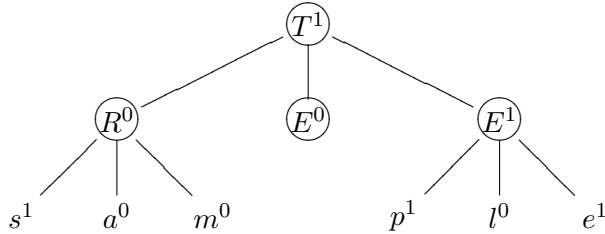
Определение 5.5. *Пусть у узла p_1 есть сыновья n_1, n_2, \dots, n_k . А у узла $n_i, (1 \leq i \leq k)$ есть сыновья c_1, c_2, \dots, c_m . Вырезанием узла n_i из дерева, называется операция замены в списке сыновей p_1 узла n_i на список своих сыновей c_1, c_2, \dots, c_m .*

Пример 5.1. *Сокращенное дерево вывода для дерева вывода из предыдущего примера.*

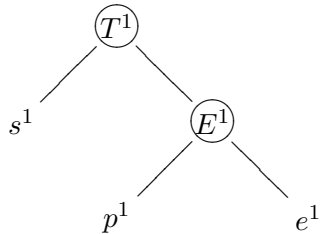
Исходное дерево вывода:



Вспомогательное дерево вывода:



Сокращенное дерево вывода:



Лемма 5.1. *Вырезание узлов при сокращении дерева можно выполнять в произвольном порядке.*

Для того, чтобы двигаться дальше, нужно ввести дополнительный термин «генератор синтаксических анализаторов».

Определение 5.6. Пусть CT – это множество всех сокращенных деревьев вывода. Определим $ct\langle G, m \rangle$ как синтаксический анализатор, который слову ставит в соответствие его сокращенное дерево вывода в грамматике G с разметкой m .

Будем обозначать ct функцию, которая размеченной грамматике ставит в соответствие синтаксический анализатор $ct\langle G, m \rangle$.

6. Безопасность AE^*

Приступим к изучению свойств функции ct .

Определение 6.1. Пусть Φ – множество расширяющих преобразований грамматик. Будем говорить, что Φ – безопасная система преобразований, если для любой размеченной грамматики (G_1, m_1) и любого преобразования $\phi \in \Phi$ существует такая разметка m_2 грамматики $G_2 = \phi(G_1)$, что выполняются следующие условия:

1. $m_1|_{P_1 \cap P_2} = m_2|_{P_1 \cap P_2}$;
2. $ct\langle G_2, m_2 \rangle|_{L(G_1)} = ct\langle G_1, m_1 \rangle$.

Другими словами, каким бы образом безопасная система преобразований не изменила грамматику, всегда удастся доопределить разметку на новых правилах так, чтобы соответствующие синтаксические анализаторы оказались совместимы. Кроме того, первое условие нам гарантирует, что новая разметка будет отличаться от старой только на новых правилах. Это означает, что изменение разметки будет носить локальный характер.

Ниже будет показано, что система преобразований AE^* является безопасной – этот факт является одним из главных результатов данной работы.

Лемма 6.1. Система преобразований AE является безопасной.

Доказательство. Возьмем произвольную размеченную грамматику (G_1, m_1) и преобразование ϕ из $AE = Add \cup Extract$. Пусть $G_1 = (V_1, \Sigma_1, P_1, S)$. Рассмотрим два случая.

Случай 1. $\phi \in Add$. Для любого слова w из языка $L(G_1)$ рассмотрим его дерево вывода в грамматике G_1 . Очевидно, это дерево также будет являться деревом вывода слова w и в грамматике $G_2 = \phi(G_1) = (V_1, \Sigma_2, P_2, S)$. Следовательно, вспомогательное дерево вывода слова w в грамматике G_1 будет также являться вспомогательным деревом вывода w в G_2 . Рассмотрим произвольную разметку m_2 грамматики G_2 такую, что $m_2|_{P_1} = m_1$. Поскольку во вспомогательном дереве вывода присутствуют лишь узлы, полученные с помощью правил вывода из P_1 , на которых разметка сохранилась, то сокращенное дерево вывода w в (G_1, m_1) совпадет с сокращенным деревом вывода w в (G_2, m_2) , то есть $ct\langle G_1, m_1 \rangle(w) = ct\langle G_2, m_2 \rangle(w)$.

Случай 2. $\phi \in Extract$. Для любого слова w из языка $L(G_1)$ рассмотрим его вспомогательное дерево вывода в грамматике G_1 . Обозначим через $Nodes$ множество всех узлов этого дерева и введем на этом множестве три функции:

1. $Sym : Nodes \rightarrow \Sigma_1 \cup V_1$ такую, что узел $n \in Nodes$ помечен символом $Sym(n)$.
2. $Sons$, которая каждому узлу $n \in Nodes$ ставит в соответствие упорядоченный кортеж узлов, являющихся сыновьями n .
3. $Prod : \{n \in Nodes | Sym(n) \in V_1\} \rightarrow P_1$ такую, что $Prod(n)$ – это правило вывода, по которому был раскрыт узел n .

Пусть ϕ удаляет правило $q_0 : A \rightarrow \alpha\beta\gamma$ и добавляет вместо него правила $q_1 : A \rightarrow \alpha B\gamma$ и $q_2 : B \rightarrow \beta$.

Для каждого элемента \bar{n} из множества $Nodes(q_0) = \{n \in Nodes | Prod(n) = q_0\} \neq \emptyset$ сделаем следующую операцию:

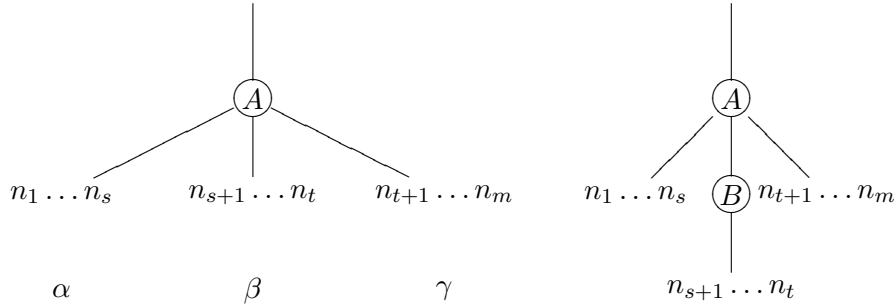
Пусть $Sons(\bar{n}) = (n_1, \dots, n_m)$. Тогда существуют такие два целых числа s, t ($0 \leq s \leq t \leq m$), что

$$Sym(n_1) \dots Sym(n_s) = \alpha$$

$$Sym(n_{s+1}) \dots Sym(n_t) = \beta$$

$$Sym(n_{t+1}) \dots Sym(n_m) = \gamma$$

Создадим новый узел \dot{n} , помеченный B , и все узлы $n_{s+1} \dots n_t$, вместе со своими поддеревьями, сохраняя порядок, сделаем сыновьями \dot{n} . А у узла \bar{n} удалим их из списка сыновей, а на их место добавим узел \dot{n} . Схематично эту операцию можно представить следующим образом:



Несложно заметить, что после того, как все узлы, раскрытые по правилу вывода q_0 , будут преобразованы описанным образом, мы получим вспомогательное дерево вывода слова w в грамматике $G_2 = \phi(G_1) = (V_2, \Sigma_1, P_2, S)$. И действительно, слово, читающееся слева направо по листьям, помеченным терминалами, не изменилось. Узлы не из $Nodes(q_0)$ мы не изменяли и можно считать, что они были раскрыты по правилам вывода из $P_1 \setminus \{q_0\} \subset P_2$. Узлы из $Nodes(q_0)$ после изменения, можно считать раскрытыми по $q_1 \in P_2$, а вновь созданные узлы, можно считать раскрытыми по $q_2 \in P_2$.

Рассмотрим произвольную разметку m_2 грамматики G_2 такую, что

1. $m_2|_{P_2 \setminus \{q_1, q_2\}} = m_1|_{P_1 \setminus \{q_0\}}$
2. $\forall i \leq s \Rightarrow m_2(q_1, i) = m_1(q_0, i)$
3. $m_2(q_1, s+1) = 0$
4. $\forall i \geq 1 \Rightarrow m_2(q_1, s+1+i) = m_1(q_0, t+i)$
5. $\forall i \Rightarrow m_2(q_2, i) = m_1(q_0, s+i)$

Рассмотрим вспомогательное дерево вывода w в G_1 и полученное из него вспомогательное дерево вывода w в G_2 . При сокращении этих двух деревьев, после добавления искусственного корня, первым делом удалим все узлы, раскрытые по q_2 (это сделать можно по лемме 5.1). Очевидно, после этого оба дерева станут равны, а разметки совпадут на всех правилах, по которым был раскрыт хотя бы один узел, оставшийся в деревьях. Но в этом случае, дальнейшее сокращение будет проходить одинаково в обоих деревьях. А это значит, что сокращенное дерево вывода w в (G_2, m_2) совпадает с сокращенным деревом вывода w в (G_1, m_1) . Другими словами $ct\langle G_1, m_1 \rangle(w) = ct\langle G_2, m_2 \rangle(w)$.

Лемма 6.2. Пусть $G \in CFG, \phi \in AE^*, w \in L(G)$. Рассмотрим вывод w в G :

$$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow w$$

Тогда существует следующий вывод w в $\phi(G)$:

$$S \Rightarrow \alpha_{0,1} \Rightarrow \alpha_{0,2} \Rightarrow \dots \Rightarrow \alpha_1 \Rightarrow \alpha_{1,1} \Rightarrow \alpha_{1,2} \Rightarrow \dots \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow w$$

Теорема 6.1. Система преобразований AE^* является безопасной.

Доказательство. Возьмем произвольную размеченную грамматику (G_1, m_1) из Γ_M и произвольное преобразование $\phi = \phi_n \circ \dots \circ \phi_2 \circ \phi_1$ из AE^* . Введем обозначения:

$$G_1 = (V_1, \Sigma_1, P_1, S);$$

$$G_{i+1} = (V_{i+1}, \Sigma_{i+1}, P_{i+1}, S) = \phi_i(G_i), i = 1..n.$$

Согласно лемме 6.1, существуют такие разметки $m_i : P_i \rightarrow \{0, 1\}, i = 2..n+1$, что

$$\begin{aligned} m_2|_{P_1 \cap P_2} &= m_1|_{P_1 \cap P_2} \\ m_3|_{P_2 \cap P_3} &= m_2|_{P_2 \cap P_3} \\ &\dots \\ m_{n+1}|_{P_n \cap P_{n+1}} &= m_n|_{P_n \cap P_{n+1}} \end{aligned}$$

При этом для любого i из промежутка от 1 до n $ct\langle G_{i+1}, m_{i+1} \rangle$ совместим с $ct\langle G_i, m_i \rangle$.

Докажем, что $ct\langle G_{n+1}, m_{n+1} \rangle$ совместим с $ct\langle G_1, m_1 \rangle$. Для этого достаточно показать, что $m_{n+1}|_{P_1 \cap P_{n+1}} = m_1|_{P_1 \cap P_{n+1}}$.

Заметим, что из определения m_i можно заключить, что $m_{n+1}|_{P_1 \cap P_2 \cap \dots \cap P_n} = m_1|_{P_1 \cap P_2 \cap \dots \cap P_n}$. Докажем, что множество $P_1 \cap P_{n+1} \setminus (P_2 \cap \dots \cap P_n)$ пусто. Очевидно, что этого достаточно для завершения доказательства теоремы.

Предположим обратное. Тогда существует правило вывода $q \in P_1 \cap P_{n+1} \setminus (P_2 \cap \dots \cap P_n)$. Обозначим $k = \max\{i | 1 < i < n+1, q \notin P_i\}$. Тогда $q \in P_{k+1} \setminus P_k$.

Найдем любое слово $w \in L(G_1)$ такое, что в его выводе в грамматике G_1 присутствует применение правила q , преобразующее α в β :

$$S \Rightarrow \dots \Rightarrow \alpha \Rightarrow \beta \Rightarrow \dots \Rightarrow w$$

По лемме 6.2 в грамматике G_k , вывод слова w будет иметь вид:

$$S \Rightarrow \dots \Rightarrow \alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_r \Rightarrow \beta \Rightarrow \dots \Rightarrow w$$

Причем $r > 0$, так как $q \notin P_k$.

По лемме 6.2 в грамматике G_{k+1} , вывод слова w будет иметь вид:

$$S \Rightarrow \dots \Rightarrow \alpha \Rightarrow \bar{\alpha}_1 \Rightarrow \bar{\alpha}_2 \Rightarrow \dots \Rightarrow \bar{\alpha}_{\bar{r}} \Rightarrow \beta \Rightarrow \dots \Rightarrow w$$

Причем $\bar{r} \geq r > 0$.

Однако поскольку $q \in P_{k+1}$, то существует также другой, отличный от данного, вывод слова w в G_{k+1} :

$$S \Rightarrow \dots \Rightarrow \alpha \Rightarrow \beta \Rightarrow \dots \Rightarrow w$$

Что противоречит однозначности грамматики G_{k+1} . А значит действительно $P_1 \cap P_{n+1} \setminus (P_2 \cap \dots \cap P_n) = \emptyset$.

7. Заключение

В качестве резюме, перечислены ключевые результаты данной работы.

Была предложена простая и выразительная система преобразований грамматик AE^* .

Были предложены концепции размеченной грамматики, синтаксического анализатора *ct* и сокращенных деревьев вывода. Синтаксический анализатор управляется размеченной грамматикой и выдает в качестве результата своей работы сокращенные деревья вывода.

Показано, что при применении к размеченной грамматике преобразований из AE^* , разметку u полученной в результате грамматики можно доопределить на новых правилах вывода так, чтобы синтаксический анализатор после применения к его грамматике преобразования из AE^* оставался совместимым с исходным синтаксическим анализатором.

Полученный результат позволяет определить аналогичные классы преобразований *Add* и *Extract* на множестве размеченных грамматик так, чтобы они изменяли разметку исходных правил вывода лишь там, где это необходимо для сохранения обратной совместимости исходного и расширенного синтаксических анализаторов.

Локальность изменения разметки при расширении понадобится, когда встанет вопрос об одновременном применении нескольких независимых преобразований одной и той же грамматики. Тот факт, что каждое из преобразований меняет разметку лишь локально, уменьшит количество конфликтов, в которых несколько преобразований претендует на изменение одних и тех же правил вывода.

Литература

1. Ахо А. Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Том 1. Синтаксический анализ. М.: Мир, 1978.

2. SHEARD T. JONES S. P. Template meta-programming for Haskell. In Proceedings of the Haskell workshop, pp. 1–16. ACM Press, 2002.
3. MOSKAL M. OLSZTA P. W. SKALSKI K. Nemerle. Introduction to a Functional .NET Language, www.nemerle.org.
4. SKALSKI K. Syntax-extending and type-reflecting macros in an object-oriented language. Master Thesis, Institute of Computer Science University of Wroclaw, 2005.
5. WILSON G. V. Extensible Programming for the 21st Century. ACM Queue, Programming languages, Vol. 2, No. 9 – Dec/Jan 2004–2005.
6. ZEMPLICKA, M. Principles of Kind Parsing. PhD Thesis. Charles Univerity, Faculty of Mathematics and Physics, Prague, Czech Republic, December 2006.