Как автоматизированно обходиться с неоднозначностями грамматик

Артём Смородский, 2 группа

20.10.2022

1) Основным недостатком общего синтаксического анализа является то, что синтаксический анализатор может создать несколько деревьев синтаксического анализа. В этом случае грамматика была не только недетерминированной, но и двусмысленной.

2) Есть в основном два вида решений для устранения двусмысленности в грамматиках. Первый включает в себя реструктуризацию грамматики, чтобы принять тот же набор предложений, но с использованием других правил. Второй оставляет грамматику как есть, но добавляет неоднозначности. Преимущество устранения неоднозначности по сравнению с реструктуризацией грамматики заключается в том, что форма правил и, следовательно, форма деревьев синтаксического анализа остается неизменной. Это позволяет языковым инженерам поддерживать предполагаемую семантическую структуру языка, сохраняя деревья синтаксического анализа, напрямую связанные с абстрактными синтаксическими деревьями.

3) Расссмотрим методы решения неоднозначностей:

a) Приоритет запрещает определенные прямые ребра между парами правил в деревьях синтаксического анализа чтобы повлиять на приоритет оператора. Формально пусть отношение приоритета $>$ является частичным порядком между рекурсивными правилами грамматики выражения. Если $A \to a_1 A a_2 > A \to \beta_1 A \beta_2$ тогда все производные $\gamma A \delta \Rightarrow \gamma(a_1 A a_2)\delta \Rightarrow \gamma(a_1(\beta_1 A \beta_2)a_2)$ являются незаконными.

b) Ассоциативность похожа на приоритет, но отец и ребенок — это одно и то же правило. Это может использоваться для воздействия на ассоциативность операторов. Левая и правая ассоциативность двойственны, а неассоциативность означает никакое вложение не допускается вообще. Формально, если рекурсивное правило $A \to AaA$ определено левоассоциативный, то любой вывод $\gamma A \delta \Rightarrow \gamma(AaA)\delta \Rightarrow \gamma(Aa(AaA))\delta$ является незаконными.

c) Резерв запрещает фиксированный набор терминалов с определенного (не)терминала, комиспользуется только для резервирования ключевых слов из идентификаторов. Пусть K будет набором предложений и пусть I нетерминал, от которого они объявлены зарезервированными. Тогда для каждого $a \in K$ любой вывод $\mathrm{I} \Rightarrow^* a$ является незаконными.

d) Динамический резерв запрещает динамический набор подпредложений из определенного не- терминал, т.е. с помощью таблицы символов. Семантика аналогична Reject, где множество K динамически изменяется в виде определенных производных (т.е. типа декларации).

e) Типы удаляют определенные поддеревья с неправильным типом с помощью средства проверки типов, оставляя правильно типизированные деревья как есть [12]. Пусть C(d) истинно тогда и только тогда, когда вывод d (представленный деревом) является корректной частью программы. Тогда все производные $\gamma A\delta \Rightarrow \gamma(a)\delta$ недопустимы, если C(a) ложно.

4) Причины неоднозначностей

Неоднозначность вызвана тем фактом, что грамматика может вывести одно и то же предложение по крайней мере двумя способами. Это не особенно интересная причина, так как она характеризует всю двусмысленность вообще. Мы заинтересованы в том, чтобы объяснить инженеру по грамматике, что не так с конкретной грамматикой и предложением, и как, возможно, решить эту конкретную проблему. Мы заинтересованы в коренных причинах конкретных вхождений узлов выбора в синтаксических лесах. Например, давайте рассмотрим конкретную грамматику языка программирования C, для которой подпредложение $\{S \cdot b;\}$ неоднозначно. В одном из производных это блок одного оператора, который умножает переменные. C a также б , в другом это блок одиночного объявления указателя переменной b на что-то типа C. C точки зрения языкового инженера, причины этого двусмысленного предложения таковы:

a) * используется как в правиле, определяющем умножение, так и в правиле, определяющем типы указателей, а также

b) Имена типов и имена переменных имеют одинаковый лексический синтаксис, а также

c) Блоки кода начинаются с возможно пустого списка объявлений и заканчиваются, возможно, пустым списком операторов, а также

d) Как утверждения, так и декларации заканчиваются на ;

Соединение всех этих причин объясняет нам, почему существует двусмысленность. Удаление только одного из них исправляет это. На самом деле мы знаем, что для C неоднозначность была устранена путем введения средства устранения неоднозначности, которое резервирует любое объявленное имя типа из имен переменных с использованием таблицы символов во время синтаксического анализа, эффективно устраняя вторую причину.

5) Неоднозначности в YACC

Множество грамматических правил неоднозначно, если существует входная цепочка, которую можно структурировать несколькими различными способами. Например, правило

$expr : expr - expr;$

естественный способ выразить тот факт, что арифметическое выражение можно сформировать, взяв два других выражения и разделив их знаком минус. К сожалению, это правило не полностью определяет способ структуризации всей совокупности исходных дан- ных. Так, если входная цепочка - это

$expr - expr - expr$

правило позволяет представить ее как в виде

$(expr - expr) - expr$

так и в виде

$expr - (expr - expr)$

YACC выявляет подобные неоднозначности при построении алгоритма разбора. На примере входной цепочки

$expr - expr - expr$

рассмотрим проблему, с которой сталкивается алгоритм. После того, как алгоритм считывает второе вхождение expr, исходная цепочка вида

$expr - expr$

сопоставляется с правой частью правила. Алгоритм может выполнить свертку с применением данного правила, после чего цепочка преобразуется в expr (левая часть правила). Затем алгоритм считывает окончание входной цепочки

$-expr$

и снова выполняет свертку. Такая последовательность действий соответствует левой ассоциативности.

Если алгоритм прочитал

$expr - expr$

он, напротив, может отложить немедленное применение правила и продолжить чтение до тех пор, пока не будет считана цепочка

$expr - expr - expr$

Алгоритм может теперь применить правило к трем правым символам и свернуть их в expr, что даст в результате

$expr - expr$

Теперь можно еще раз выполнить свертку. Такая последовательность действий соответствует правой ассоциативности. Итак, прочитав

$expr - expr$

алгоритм может сделать один из двух шагов, каждый из которых законен: перенос или свертку. Нет оснований предпочесть один из них. Такая ситуация называется конфликтом перенос-свертка. Может произойти и так, что алгоритм будет иметь выбор между двумя допустимыми свертками. Такой конфликт называется конфликтом свертка-свертка. Отметим, что конфликтов перенос-перенос возникнуть не может.

YACC по умолчанию использует два метода разрешения неоднозначностей:

При конфликте перенос-свертка по умолчанию выполняется перенос.

При конфликте свертка-свертка по умолчанию выбирается правило, встретившееся в yacc-спецификации первым.

Первый метод устанавливает, что свертки менее предпочтительны, чем переносы, если есть выбор. Второй метод дает пользователю довольно грубый механизм управления разбором, поэтому, когда возможно, конфликтов свертка-свертка следует избегать.

6) Неоднозначности в Antlr

В естественном языке существуют неоднозначно трактуемые фразы. В формальных языках подобные конструкции также могут встречаться. Например, следующий фрагмент:

stat: expr ';' // expression statement

| ID '(' ')' ';' // function call statement;

;

expr: ID '(' ')'

| INT

;

Однако, в отличие от естественных языков, они скорее всего являются следствием неправильно разработанных грамматик. ANTLR не в состоянии обнаруживать такие неоднозначности в процессе генерации парсера, но может обнаруживать их непосредственно в процессе парсинга, если устанавливать опцию $LL\_EXACT\_AMBIG\_DETECTION$. Неоднозначность может возникать как в лексере, так и в парсере. В лексере для двух одинаковых лексем, формируется токен, объявленный выше в файле (пример с идентификаторами). Однако в языках, где неоднозначность действительно допустима (например, C++), можно использовать семантические предикаты (вставки кода) для ее разрешения, например:

expr: isfunc(ID) ? ID '(' expr ')' // func call with 1 arg

| istype(ID) ? ID '(' expr ')' // ctor-style type cast of expr

| INT

| void

;

Также иногда неоднозначность можно исправить с помощью небольшого переделывания грамматики. Например, в C существует оператор побитового сдвига вправо RIGHT_SHIFT: >>; две угловые скобки могут также использоваться для описания классов-генериков: $List < List < int >>$. Если определить токен >>, то конструкция из двух списков никогда не сможет распарситься, потому что парсер будет считать, что вместо двух закрывающихся скобок написан оператор >>. Чтобы решить такую проблему, достаточно просто отказаться от токена RIGHT_SHIFT. При этом токен LEFT_SHIFT: << можно оставить, поскольку такая последовательность символов при парсинге угловых скобок не может встретиться в валидном коде.

7) Ссылки:

http://cable.pu.ru/info/unix_c/p13.html04

https://habr.com/ru/company/pt/blog/210772/?ysclid=l9j3luk65j691438830

https://homepages.cwi.nl/ jurgenv/papers/SLE2011-2.pdf